Reasoning about Correctness Properties of a Coordination Programming Language

by Gudmund Grov

Submitted for the Degree of
Doctor of Philosophy
at Heriot-Watt University
on Completion of Research in the
School of Mathematical and Computer Sciences
March 2009

Abstract

Safety critical systems place additional requirements to the programming language used to implement them with respect to traditional environments. Examples of features that influence the suitability of a programming language in such environments include *complexity of definitions*, *expressive power*, *bounded space and time* and *verifiability*. Hume is a novel programming language with a design which targets the first three of these, in some ways, contradictory features: fully expressive languages cannot guarantee bounds on time and space, and low-level languages which can guarantee space and time bounds are often complex and thus error-phrone. In Hume, this contradiction is solved by a two layered architecture: a high-level fully expressive language, is built on top of a low-level coordination language which can guarantee space and time bounds.

This thesis explores the *verification* of Hume programs. It targets *safety* properties, which are the most important type of correctness properties, of the low-level coordination language, which is believed to be the most error-prone. Deductive verification in Lamport's *temporal logic of actions* (TLA) is utilised, in turn validated through algorithmic experiments. This deductive verification is mechanised by first embedding TLA in the *Isabelle* theorem prover, and then embedding Hume on top of this. Verification of temporal invariants is explored in this setting.

In Hume, program transformation is a key feature, often required to guarantee space and time bounds of high-level constructs. Verification of transformations is thus an integral part of this thesis. The work with both invariant verification, and in particular, transformation verification, has pinpointed several weaknesses of the Hume language. Motivated and influenced by this, an extension to Hume, called *Hierarchical Hume*, is developed and embedded in TLA. Several case studies of transformation and invariant verification of Hierarchical Hume in Isabelle are conducted, and an approach towards a calculus for transformations is examined.

Acknowledgements

First of all, I would like to thank my supervisors, Andrew Ireland and Greg Michaelson, who have given me help, support and guidance throughout this thesis. I would also like to thank my external examiner, Ursula Martin, and my internal examiner, Lilia Georgieva, for the encouraging feedback they gave me. I have also received help from other sources, in particular: Stephan Merz; Robert Pointon; Kevin Hammond; Hans-Wolfgang Loidl; Lucas Dixon; Steffen Jost; Jörg Siekmann; the members of the EmBounded project; the members of the Dependable System Group at the School of Mathematical and Computer Sciences (MACS) at Heriot-Watt University; the members of the Mathematical Reasoning Group at Edinburgh University; the staff within MACS; and all the anonymous reviewers that gave constructive feedback on my submitted papers.

I would also like to acknowledge all the support and friendship received from my friends and colleagues over the years. I owe dept and gratitude to my family: my sister Kjersti; my sister Anja and her family Michael and Sebastian; and especially my parents, Elsa and Rune. Finally, special thanks goes to Ellen, who supported and helped when I needed it the most, and even proof read the whole thesis.

Financially, I have been gratefully supported by the following sources, without which I would not have been able to complete this thesis: a James Watt Scholarship from Heriot-Watt University; Johan Helmich Janson og Marcia Jansons legat; Lise og Arnfinn Hejes fond; Knut Hamsuns minnefond; Ludvig Daae Løvestad legat; direktør Halvor B. Holtas legat ved NTNU; Nansenfondet og de dermed forbundne fond; Petter Dass stipendiefond; EU FP6 EmBounded project; EPSRC Platform grant GR/S01771 (The Integration and Interaction of Multiple Mathematical Reasoning Processes); and the Norwegian state loan fund.

Contents

Chapter 1

Introduction

1.1 Motivations

In the relative short history of computers, there are many cases where errors have resulted in both financial and human costs. The following examples are taken from Wired [74] and DevTopics [55]. In the *Mariner 1 space probe* (1962), a computer bug caused the rocket to divert from its intended launch path, forcing mission control to destroy it over the Atlantic ocean. The error had an estimated cost of $18.5 million (at the 1962 value). Due to a race-condition bug in the Canadian *Therac-25 medical accelerator*, at least 5 people were killed, and others seriously injured (1985). The reuse of code from Ariane 4 on a different architecture caused *Ariane 5 flight 501* to crash 40 seconds after launch (1996). And, a bug in a Soviets warning software indicated that the US had launched five ballistic missiles, almost starting a third world war in 1983. In more recent times, the Financial Times [110] reported that a bug in *Moody's* software caused it to incorrectly award triple-A ratings, thus mistakenly accepting sub-prime mortgages worth billions of dollars. Problems with such sub-prime mortgages were part of the reasons behind the current *global credit crunch*.

The software industry now uses at least 30% of its budget on verification and testing, and Gartner Inc [101] estimates that the worldwide testing market will reach $13 Billion by 2010. However, for safety, security and mission-critical systems like the ones described above, verification and testing count for closer to 80% of the total budget [14].

In [180], Storey identifies *complexity of definitions*, *expressive power*, *bounded space and time*, *logical soundness*, *security* and *verifiability* as the key features for a programming language targeting safety-critical systems. No language supports all of these features, and several of them are conflicting: for example, results from Turing [185] and Gödel [78] show that the verification problem is, in general, undecidable for sufficiently

expressive programming languages, called *Turing-complete* languages.

Memory leaks are common errors in computer programs, hence the inclusion of space bounds in Storey's list. Clearly, this becomes an even more important property in resource constrained environments, like embedded systems. These are often control systems, where response time is another key feature. However, determining time and space bounds are undecidable for Turing-complete languages, while languages where time and space properties are decidable, are often not expressive enough to work with effectively, and tend to be too low level.

Hume [92] is a Turing-complete programming language which attempts to guarantee *bounded space and time*, while being high-level, meaning a *low complexity of definitions*. This is achieved by a *layered* design, where a finite state automata *coordination layer* with a large set of decidable properties, is built on top of a Turing-complete declarative *expression layer*. Programming involves balancing the expressiveness and high-level of the expression layer, with decidable properties of the coordination layer. Thus, Hume is divided into a set of *levels*, where each level has a set of allowed constructs and decidable properties. Hume can therefore be seen as a family of languages.

The resource analysis required to guarantee bounded space and time is currently only well-developed for the expression layer. Moreover, many environments targeted by Hume have a strong requirement for *correctness*. One example is the UK Ministry of Defence standard [152], which describes the requirements for software developed for the UK MoD. Currently, Hume cannot give any correctness guarantees. This thesis attempts to fill this gap, by exploring *formal correctness verification of Hume programs*.

Formal verification is a form of *formal methods*, a notion used in the application of mathematical tools and techniques in the development of software and hardware systems. In addition to *formal verification*, *formal specification* and *formal development* are considered formal methods, and these features are often connected: for example, a property must be specified to be verified. Formal methods are mainly applied when there are strong safety or security requirements, where merely testing is not sufficient, since only selected cases can be checked, and *not all cases*. The rigour of the application of formal methods, depends on the nature of the problem, and in most cases it is only applied to the safety-critical parts of the program, not the complete program. Rushby [174] has thus identified four levels of rigour labelled from 0 to 3: level 0 is no use of formal methods; level 1 is the use of concepts and notation from discrete mathematics; level 2 is the use of formalised specification languages with some mechanised support tools; while level 3 is full use of formal specification and mechanisation of proofs. Mechanised tool support consists either of an automatic algorithmic approach known as *model checking*, or a more expressive deductive approach known as *theorem proving*,

which often requires user interaction.

Although no evidence is provided, the claim that verification of the Hume coordination layer is more challenging than the expression layer due to its low-levelness, should be uncontroversial. Thus, this thesis focuses on the coordination layer. Moreover, level 3 rigour is the target, albeit only a subset of the language and properties is discussed.

The Hume methodology is based around decidability analysis and transformations. A high level program is first developed. If space and time bounds cannot be guaranteed, it is transformed into a lower, more decidable level. Thus, since program transformations is such an integral part of the methodology, transformation verification is the other main topic of this thesis. Finally, although *software verification* is the main topic, the two other "types" of formal methods: *specification* of properties, and *development* in the form of program transformations, are discussed.

In order to formally verify Hume programs, they must be represented in a formal logic, preferable with mechanical tool support. To enable both the reuse of current tools and techniques, and to support the reuse of tools and techniques developed in this thesis, the use of an existing logic is a target. When compared with developing a purpose-specific logic for Hume, the use of an existing logic liberates one from dealing with low-level logical details like a soundness proof of the logical system. Leslie Lamport's novel *temporal logic of actions* (TLA) [120], is the main logic used here. The Hume coordination layer can be seen as a concurrent system, and TLA is a result of more than 30 years of experience with such systems. Moreover, the structure of TLA fits very well with both the structure of Hume and the target of this thesis, and includes tool support via the TLC model checker [124, 203].

This thesis is the first step towards a verification environment for Hume programs. Thus, extensibility of the tools and techniques discussed here is a key feature. In particular, the expression layer of Hume is being developed in parallel in Isabelle/HOL [135]. Hence, to enable future integration with this work, the focus is on theorem proving in Isabelle/HOL, compared to develop TLC for Hume.

1.2 Contributions

The main contributions of this thesis are:

- correctness and transformation verification of Hume programs are discussed;
- TLA is used at the programming language level and in program transformation verification;

- TLA is mechanised in the Isabelle/HOL theorem prover (Isabelle/TLA) in particular, through the mechanisation of sequences for "TLA style" logics;
- Hume is formalised in TLA. Hume is mechanised in Isabelle/TLA and the TLC model checker. Several new tactics which automate Hume reasoning in Isabelle/TLA are developed;
- an extension to Hume, called Hierarchical Hume, is formalised in TLA and mechanised in Isabelle/TLA. Tactics that automate invariant and transformation verification are developed for Hierarchical Hume;
- the Hierarchical Hume extension and an existing scheduling extension, called self-out scheduling, are formally verified to be conservative using TLA;
- the approaches are illustrated by case studies, among them an implementation of NASA's SAFER system in Hierarchical Hume, and real-time model checking of Hume using TLC;
- a box calculus for Hume transformations is outlined.

Chapter 9 surveys other more experimental contributions:

- the mechanisation of Hume in Isabelle/TLA is integrated with a mechanisation of the expression layer [135], which is developed in parallel in Isabelle/HOL. The integration is within Isabelle/HOL, and applied to an example;
- the use of rippling to automate both correctness and transformation proofs is discussed;
- the liveness of Hume programs is discussed, and is illustrated by model checking.

Publications

- Gudmund Grov. Verifying the Correctness of Hume Program – An Approach Combining Algorithmic and Deductive Reasoning. *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE-05)*, pages 444 – 447. ACM Press, 2005. [85]
- Kevin Hammond, Gudmund Grov, Greg Michaelson, and Andrew Ireland. Low-level programming in Hume: an exploration of the HW-Hume level. In Zoltán Horváth, Viktória Zsók, and Andrew Buttereld, editors, *In Proceedings of Implementation of Functional Languages (IFL 2006)*, volume 4449 of Lecture Notes in Computer Science, pages 91 – 107. Springer, 2006. [91]

- Gudmund Grov and Greg Michaelson. Towards a Box Calculus for Hierarchical Hume. In Marco T. Morazan, editor, *Trends in Functional Programming*, volume 8, chapter 5, pages 71 – 88. Intellect, 2007. [87]
- Gudmund Grov, Greg Michaelson, and Andrew Ireland. Formal Verification of Concurrent Scheduling Strategies using TLA. *In 3rd IEEE International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, number CFP07036-USB in IEEE Catalog Number. IEEE, 2007. [88]
- Gudmund Grov, Robert Pointon, Greg Michaelson, and Andrew Ireland. Preserving Coordination Properties when Transforming Concurrent System Components. *In Coordination Models, Languages and Applications Track of the 23rd Annual ACM Symposium on Applied Computing*, volume 1 of 3, pages 126 – 127, 1515 Broadway, New York, March 2008. The Association for Computing Machinery, Inc. [89]
- Gudmund Grov and Andrew Ireland. Towards Automated Property Discovery within Hume. *In 2nd International Workshop on Invariant Generation (WING'09)*. [86]

1.3 Thesis roadmap and outline

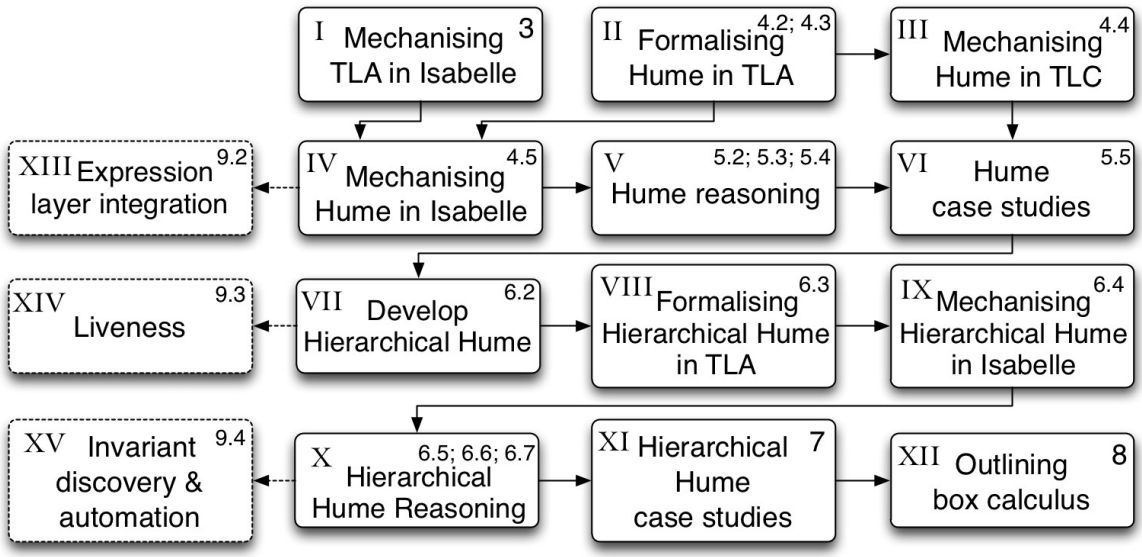


Figure 1.1: Thesis roadmap

Figure 1.1 shows the roadmap of the thesis. It has been divided up into fifteen distinct parts, where the last three, shown on left of the figure, are some relevant explorations

discussed in Chapter 9. Since these are not fully implemented and not in the critical path of the thesis, the boxes are stippled. The arrows show the dependency. Note that the arrows are assumed to be "transitive". For example, VIII depends on II. Since, VIII depends on VII, which in turn depends on VI, which depends on V and so on, the arrow from II to VIII is omitted. In the top right corner of each box, it is indicated which section or chapter implements the part. The thesis is organised as follows:

- Chapter 2 contains the relevant background information on TLA;
- Chapter 3 describes Isabelle/TLA, a mechanisation of TLA in Isabelle/HOL (Part I);
- Chapter 4 describes the Hume formalisation in TLA, the Isabelle/Hume mechanisation and Hume mechanisation in TLC (Parts II, III and IV);
- Chapter 5 describes the reasoning process of Hume within TLA, including case-studies (Parts V and VI);
- Chapter 6 introduces, formalises and mechanises Hierarchical Hume, and discusses reasoning within Hierarchical Hume (Parts VII, VIII, IX and X) ;
- Chapter 7 contains Hierarchical Hume case studies (Part XI);
- Chapter 8 outlines a box calculus for Hierarchical Hume transformations (Part XII);
- Chapter 9 discusses some relevant explorations (Parts XIII, XIV and XV);
- and finally, Chapter 10 outlines future directions and concludes.

1.4 Notation and notions

TLA, TLA$^+$, Isabelle and Hume use different syntax in relevant papers. To avoid bewildering readers who are familiar with these, an attempt is made to use the standard syntax. Thus, several different notations are used throughout the thesis, and these are summarised below:

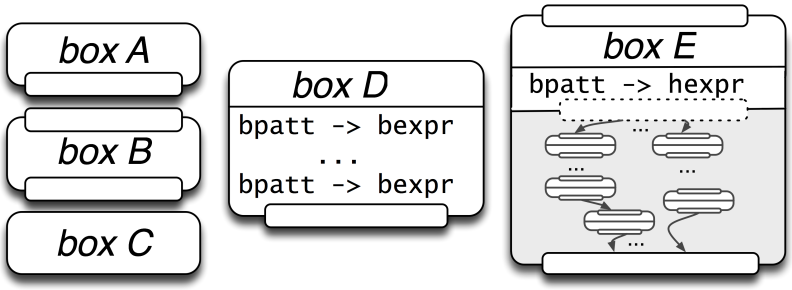
TLA terms are written in *italic* and keywords are written with a **bold font**. It uses $\triangleq$ for definitions, while $\Rightarrow$, $\wedge$, $\vee$, $\neg$ and $\equiv$ represent implication, conjunction, disjunction, negation and equivalence. $\forall$ and $\exists$ are used for quantification of constants, while $\mathbf{\exists}$ is existential quantification of variables. Reasoning with $\mathbf{\exists}$ with respect to refinements, involves finding a *refinement mapping*, which is the sum

of all witnesses of $\exists$-bound variables. For a formula F, $F[f/x]$ is the substitution of witness f for variable x in F, and is often written $\overline{F}$. Function parameters are bracketed by $(\cdots)$, and tuples are written $\langle\cdots\rangle$. ;

TLA$^+$ deviates from TLA by using THIS FONT for keywords. Moreover, functions are separated by operators. Here, function parameters are encapsulated by $[\cdots]$, whilst operator parameters are encapsulated by $(\cdots)$;

Isabelle terms are written with this font, while **this font** is used for keywords. Isabelle contains a meta-level logic and an object-level logic. In the meta-level, $\bigwedge$ and $\Longrightarrow$ are universal quantification and implication, while $\equiv$ is used for definitions. The object level logic used in this thesis is higher order logic (HOL), called Isabelle/HOL. The mechanisation of TLA in Isabelle/HOL, described in Chapter 3, is written *Isabelle/TLA*, although is should strictly speaking be Isabelle/HOL/TLA. Object-level implication is written $\longrightarrow$ and type application is written $\Rightarrow$. The remaining predicate connectives are the same as in TLA. Currying is used, thus $f(a, b)$ is written `f a b`. Tuples are written $(\cdots)$, while existential variable quantification ($\exists$) in Isabelle/TLA is here written $\exists\exists$;

Hume. All Hume code is written in `this font`. However, in most cases only a graphical representationof Hume programs, illustrated below, is shown. Currying is also here used for functions. *Isabelle/Hume* and *Isabelle/HHume* are used for the mechanisations of Hume and Hierarchical Hume in Isabelle/TLA. The following graphical notations are used for Hume boxes:



box `A` is a "standard" Hume box which buffers the output; box `B` also buffers the input, while box `C` buffer neither the input nor the output; box `D` is a "standard" flat box, which also shows the expression layer (but ignores types); finally, box `E` is a hierarchical box, which buffers the inputs and outputs. Here, $\circ\!\!\rightarrow$ is used to graphically illustrates a program transformation. The term *spatial component* is used for a cluster of boxes, which is neither semantically nor syntactically, separated from the rest of the program.

A construct is *formalised* when it is represented in a formal logic, whereas *mechanised* is used for a formalisation inside a mechanical tool, such as Isabelle. Similarly, a *formal proof* is a pen-and-paper proof in a formal logic, while a *mechanised proof* has been checked by a theorem prover. An *informal proof* is not inside a formal logic, and a *proof outline* is a high level description of a formal, informal or mechanised proof. A *program* is a (computable) algorithm that has been implemented in a programming language, and *software verification* denotes verification at the programming language level, while *assertion* and *specification* captures such requirements. *Static analysis* is analysis applied without running a program, while *dynamic analysis* is analysis while running a program. An *interpreter* runs a program inside another program, while a *compiler* turns a program into executable machine code.

Chapter 2

Background

2.1 Software verification

The idea of mathematically reasoning about programs goes back to Goldstine and von Neumann [79] and Turing [186]. However, the introduction of *Hoare-triples*, or *Hoare logic*, by Hoare [95], based on a paper by Floyd [70] is often considered the first seminal work in this area. Here, a program is described axiomatically, thus operational reasoning is not required. This is achieved by annotating program code C with pre-condition P and post-condition Q, written as

$$\{P\}C\{Q\}.$$

The triples show *partial correctness*: if P holds, and command C terminates, then Q holds afterwards. *Total correctness* is partial correctness with an additional termination proof of C. The task of finding both the code and annotation requires user interaction, but heuristics exist to guide this, such as *predicate transformers* like Dijkstra's *weakest preconditions* [56]: $wp(C, Q)$ computes the weakest precondition (state) from code C where the post-condition Q holds after execution. Weakest preconditions also handle termination.

The concept of Hoare triples is an example of *ad-hoc* verification: a given program is annotated and these annotations are then verified, while heuristics, like the weakest precondition aids the user in finding annotations. However, for non-trivial sized programs such *ad-hoc* verification is only possible for correct programs. As described in [109], *decomposition* and *refinement* are required: decomposition enables the proof of a component to be reduced to the proof of its sub-components; refinement is a result of a step-wise design: parts of the program are developed and verified gradually. The early phases are very abstract, while the later phases are closer to an implementation. One

of the first such refinement-based development methods was the Vienna Development Method (VDM) [107]. A VDM model defines a class of states and a series of operations that depend upon, and update the state, with clear semantics of de-composition and refinement [109]. A similar system to VDM is Z [179], originally proposed by Jean-Raymond Abrial. ZRC [40] is a refinement calculus for Z specifications. Based mainly on Z, Abrial developed the B-method [8], geared more towards software development and refinement. Event-B [9] is developed from B, and focuses on system development. TLA also reflects *decomposition* [4, 5] and *refinement* [3].

The Hoare-triples assume that the program C is executed sequentially in isolation, and sequences are handled by the sequential composition axiom:

$$\frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1\ ;\ C_2\ \{Q\}}.$$

However, if C_1 and C_2 are not executed in isolation, other programs may update the state at the same time as C_1 and C_2, thus $\{P\}\ C_1\ ;\ C_2\ \{Q\}$ cannot be derived due to the potential side-effect introduced by other programs. Thus, the composition axiom is not valid. In [163], Susan Owicki, supervised by David Gries, extended the Hoare-triples into concurrent systems, by dividing the proof process into two phases: first the proof is as in the sequential case; followed by a proof that the different processes could not interfere with each other. This method is known as the *Owicki-Gries method*. However, the second step requires reasoning about the complete system, thus local reasoning and decomposition are not supported, and details admitted later in the process may interfere, thus refinement is not possible.

More recently, *separation logic* [170], has been suggested for reasoning about concurrent programs [161]. Separation logic was originally developed by Reynolds [170] to reason about heap data-structures like pointers, by adding connectives to split the heap into parts. This enables local reasoning, and can be applied to concurrent system as later shown by O'Hearn [161].

Both separation logic and the Owicki-Gries method rely on an underlying state, and communication is by shared variables. Another technique for the verification of concurrent systems is the use of *process algebras* [18]. Here, communication is by processes and no variables are shared. The word 'process' refers to the behaviour of a system, while 'algebra' reflects the use of algebraic laws to manipulate and analyse the system. Hoare's CSP [96] is an example of a process algebra. It was developed as a programming language for concurrent systems. Another example is Milner's CCS [149], although its intended use was not as a programming language. The Π-calculus [150] extends CCS by allowing configurations to change during computation, while

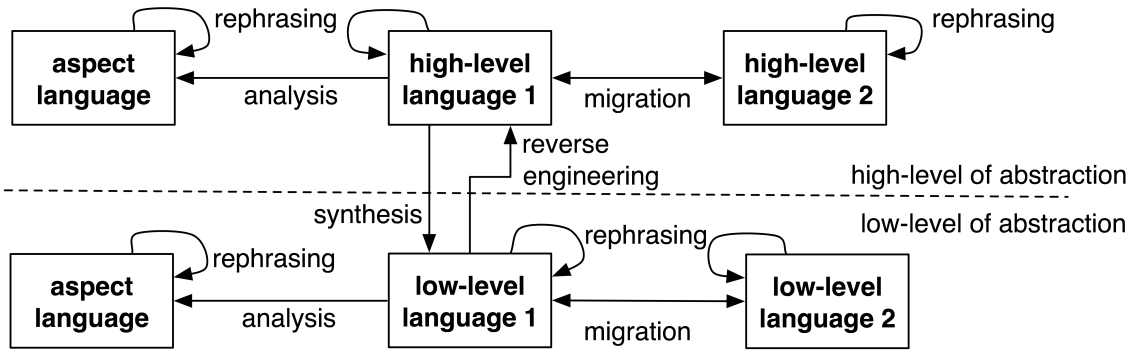


Figure 2.1: Types of program transformations [191]

the Join calculus [71] targets distributed programming languages, where rendezvous communications are not supported. Process algebras support equivalence verification of processes, which is stronger than than refinement, through the use of bisimulations.

Finally, property *specification* is an interesting topic for concurrent systems: in the sequential cases properties are specified in a classical predicate logic. For concurrent systems, *temporal logic* has been successful. It was first suggested by Burstall [35] for program development, while Pnueli [167][1] first suggested it for concurrent systems. Temporal logic permits reasoning with time without explicitly introducing it. Leslie Lamport was the first to distinguish the notions of *safety* and *liveness* [118]: a safety property asserts that bad things never occur, while a liveness property asserts that something good will eventually occur. This was later formalised by Alpern and Schneider [13]. They also showed that all properties are combination of safety and liveness properties. In general, safety properties are considered more important, and liveness properties are harder to verify. TLA is a temporal logic supporting safety, liveness, refinement and decomposition, discussed in detail in Section 2.3. This is preceded by a classification of program transformations.

2.2 Program transformation classification

Visser [191] separates between *high-level programming languages* and *low-level programming languages*: a high-level programming language abstracts over the underlying hardware; while a low-level programming language depends on the hardware. An *aspect language* is sub-language, which can be either high-level or low-level, which only shows particular aspects of the language, like the control-flow or the data-flow.

Visser classifies *program transformations* into two distinct types: *translation* and

[1]The citation is the journal version (1981) based on a report from Tel-Aviv University from 1977.

rephrasing – each of which have different sub-types. The relationship between the different sub-types is shown graphically in Figure 2.1.

Translation. In a program translation the source language deviates from the target language. *Program synthesis* is an example of a program translation, where the abstraction level is lowered. For example: *program refinement* is, as described above, an implementation derived from a high-level specification; while, in a *compilation*, the target language is machine code. The opposite of program synthesis is *reverse engineering* where the abstraction level is increased. In *program migration* the translation is between two different languages at the same level, while *program analysis* reduces a program to just one aspect. For example, in data-flow analysis only the data-flow aspect is required.

Rephrasing. In *program rephrasing* the source and target language is the same. *Normalisation* reduces a program to a program in a sub-language: *de-sugaring* eliminates syntactic sugar; *simplification* is more general, and reduces the program to a normal form, but does not need to remove simplified parts. *Program optimisation* is a transformation that improves certain properties, such as time or space usage. *Program refactoring* restructures the program design to make it easier to understand, whilst preserving the functionality: *obfuscation* transformation makes the program harder to understand for protection against re-engineering. *Program reflection* changes the program to also compute meta-values of itself, such as tracing a variable. *Program (software) renovation* changes the behaviour of a program to repair a bug, such as the Y2K bug.

2.3 The temporal logic of actions

> *"... temporal logic is a necessary evil that should be avoided as much as possible."*
>
> – Leslie Lamport [120, page 46]

The *temporal logic of actions* (TLA) targets concurrent systems. It allows both *liveness* and *safety* properties to be expressed in the same uniform logic.

A property Φ holds in a specification Π if Π *implements* Φ. These are expressed in the same uniform logic, thus logically they are not distinguished. Moreover, a specification Π refines another specification Φ, if Π *implements* Φ. Implementation is expressed as logical implication, thus both property and refinement verification are

expressed as

$$\Pi \Rightarrow \Phi. \tag{2.1}$$

TLA combines a *linear temporal logic* with an *action logic*. In a *linear* temporal logic, time is linear, compared to *branching* temporal logic, where time has a tree-like structure with a branch for each potential future direction. There, quantification over the different branches is supported, which does not hold for linear temporal logics. The temporal and action logic combination in TLA creates a three tier logic where

- in the *state level*, a *state function/predicate* is a function/predicate on one particular state. In [127], Lamport argues that type systems are hard to get right, and getting them wrong can lead to inconsistencies. He then states that set theory can be used as a basis for a type free specification language. Thus, TLA is a type free logic, where all values are members of an infinite set Val, and all variable names are members of an inifinite set Var. A *state* s assigns values to variables, i.e. it is a function from Var to Val. The value of a variable x in s is semantically $s(x)$, albeit abbreviated by x. The state level also contains a full predicate calculus;

- in the *action level*, an *action* $\mathcal{A}$ is a predicate on two states (s, t): a 'before state' s and 'result state' t state of $\mathcal{A}$. Syntactically, $s(x)$ is written x, while $t(x)$ is written x', and $'$ distributes over all operators;

- in the *temporal level*, a *formula* is a predicate on an infinite sequence of states. A sequence σ is represented as a function from natural numbers nat to states St, where St is an infinite set of all possible states. Below $\langle s_0, s_1, \cdots \rangle$ is used for σ, and $\langle t_0, t_1, \cdots \rangle$ for the sequence τ. $\sigma|_i$ is the suffix of sequence σ, starting at state s_i, i.e. $\langle s_i, s_{i+1}, \cdots \rangle$.

  Lamport uses the term *rigid variables* for what a programmer calls constants, and *flexible variables* for what are simply known as variables. $\Box$, meaning something always holds, and $\Diamond$, meaning something will eventually hold, are supported, together with an existential quantifier $\exists$ for flexible variables, which is essential for most refinement proofs. The term *behaviour* is often used for the temporal level, and St^∞ is an infinite set of all behaviours of all states in St.

A *stuttering step* is a step in a sequence that leaves the state unchanged. For example, assume that $s_0 \neq s_1$. Then the sequence $\langle s_0, s_0, s_1 \cdots \rangle$, contains a stuttering step followed by a non-stuttering step. Two sequences are *equal up to stuttering* iff they are separated only by stuttering steps. For example, $\langle s_0, s_0, s_1, s_1, s_1, s_2, \cdots \rangle$ and

$\langle s_0, s_1, s_2, \cdots \rangle$ are equal up to stuttering. To enable refinement, a formula must be *stuttering invariant*: the validity of a formula is the same for sequences that are equal up to stuttering – or steps that leave the state unchanged do not change the validity of a formula. This excludes the temporal 'next' (state) operator, which is not stuttering invariant. This can be shown by the following example. Assume that $x = 1$ holds in state s_0, written $s_0 \vDash (x = 1)$, and $x = 2$ holds in state s_1, written $s_1 \vDash (x = 2)$. Then $\langle s_0, s_1, s_2, \cdots \rangle \vDash \Big((x = 1) \Rightarrow \text{'next'}(x > 1)\Big)$ holds since anything prefixed by 'next' refers to the second state s_1 of the sequence, while non-prefixed terms refer to the first state s_0. Thus, the proof reduces to showing that $x > 1$ from $x = 2$, which holds. However, this property is invalid under sequence $\langle s_0, s_0, s_1, s_1, s_1, s_2, \cdots \rangle$, which equals $\langle s_0, s_1, s_2, \cdots \rangle$ up to stuttering. Here, $\langle s_0, s_0, s_1, s_1, s_1, s_2, \cdots \rangle \vDash \Big((x = 1) \Rightarrow \text{'next'}(x > 1)\Big)$ is invalid, since the second state, which 'next' refers to, is s_0. Here, $x = 1$ is assumed, which contradicts the required goal $x > 1$.

In TLA, computation is modelled by an action $\mathcal{A}$. The computation should be valid throughout execution (and not just in the first two steps), which requires a lifting into the temporal level. A naive lifting of $\mathcal{A}$, induces $\Box\mathcal{A}$. However, an action is not necessarily stuttering invariant, and $\Box$ only preserves, and does not introduce, stuttering invariance. Instead of $\mathcal{A}$, $\mathcal{A} \vee v' = v$ is used, where v is a tuple of state functions. This is stuttering invariant, and is abbreviated by $[\mathcal{A}]_v$. $\Box[\mathcal{A}]_v$ lifts this into into the temporal level.

All variables must be given an initial value by a state predicate I. Moreover, internal variables i of the specification can be hidden by binding them with the $\exists$ operator. Let $v = \langle e, i \rangle$ be a tuple of the external (visible) variables e and internal (hidden) variables i. In a monolithic specification, $\langle e, i \rangle$ should contain all variables in the state space, and $\mathcal{A}$ should update all variable. Such monolithic specifications are of the form:

$$\exists i.\ I \wedge \Box[\mathcal{A}]_{\langle e,i \rangle}. \tag{2.2}$$

$\exists$ hides the internal variables i, thus e are the only free variables of (2.2). I must specify the initial value of e and i, while $\mathcal{A}$ describes how e and i are updated in the computation.

2.3.1 Liveness

Only safety requirement can be verified from (2.2). To verify liveness properties, the actions must be constrained by liveness assumptions. Arbitrary liveness properties may introduce unexpected safety properties, thus the liveness assumption should be restricted to a form of liveness called *fairness*. There is a separation between *weak*

and *strong* fairness properties of actions. These require some auxiliary definitions and meanings: $\Diamond F$ denotes $\Diamond F \equiv \neg\Box\neg F$; the action variant of this is written $\Diamond\langle\mathcal{A}\rangle_v$ and is defined as $\Diamond\langle\mathcal{A}\rangle_v \triangleq \neg\Box[\neg\mathcal{A}]_v$ and it can easily be shown that $\Diamond\langle\mathcal{A}\rangle_v \equiv \Diamond(\mathcal{A} \wedge v' \neq v)$; and finally, an action $\mathcal{A}$ is *enabled* in a state s, written *Enabled* $\mathcal{A}$, if it is possible to execute $\mathcal{A}$ starting in s. Weak fairness (WF) for $\mathcal{A}$ asserts that if $\mathcal{A}$ is continously enabled then it must always eventually be executed. Strong fairness (SF) asserts that if $\mathcal{A}$ is infinitely often enabled than it must always eventually be executed:

$$\begin{array}{ll} WF_v(\mathcal{A}) & \triangleq \Diamond\Box \mathit{Enabled}\,\langle\mathcal{A}\rangle_v \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v \\ SF_v(\mathcal{A}) & \triangleq \Box\Diamond \mathit{Enabled}\,\langle\mathcal{A}\rangle_v \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v \end{array} \tag{2.3}$$

Now, (2.2) can be extended with fairness as follows: $\exists i.I \wedge \Box[\mathcal{A}]_{\langle v,i\rangle} \wedge \mathit{Live}$, where *Live* is a conjunction of *SF* and *WF* declarations. If all actions used in the *WF/SF* declarations are sub-actions of $\mathcal{A}$, then no unexpected safety properties have been introduced and the specification is *machine closed* [5, page 519].

2.3.2 TLA semantics

A detailed exploration of the logical details of TLA is discussed in [7], while this discussion is based on [120]. The following definitions show the semantics for the safety and propositional parts of the TLA semantics:

(1)	$s \vDash f$	*iff*	$f[s(v)/v]$ *holds for all variables* v
(2)	$(s,t) \vDash \mathcal{A}$	*iff*	$\mathcal{A}[s(v)/v, t(v)/v']$ *holds for all variables* v
(3)	$\sigma \vDash \neg A$	*iff*	$\sigma \vDash A$ *does not hold.*
(4)	$\sigma \vDash A \wedge B$	*iff*	$\sigma \vDash A$ *and* $\sigma \vDash B$ *holds.*
(5)	$\vDash \mathcal{A}$	*iff*	$(s,t) \vDash \mathcal{A}$ *holds for all* $s, t \in \mathbf{St}$
(6)	$\vDash F$	*iff*	$\forall\sigma \in \mathbf{St}^\infty.\ \sigma \vDash F$
(7)	$\sigma \vDash \Box F$	*iff*	$\sigma|_i \vDash F$ *holds for all* $i \geq 0$.
(8)	$\sigma \vDash \mathcal{A}$	*iff*	$(s_0, s_1) \vDash \mathcal{A}$.

(1) $f[s(v)/v]$ denotes v substituted by $s(v)$ in f. Thus, (1) implies a state predicate f of state s holds if all variables are substituted by their (semantic) values. (2) Similarly, primed variables are looked up in the after state t and unprimed in the before state s of an action $\mathcal{A}$. (3) A behaviour satisfies $\neg A$ iff it does not satisfy A. (4) A behaviour satisfies $A \wedge B$ iff it satisfies A and it satisfies B. The remaining propositional operators can be derived from these. (5) An action is valid iff it holds for all state pairs, while (6) a formula is valid iff it holds for all sequences. (7) A formula always holds ($\Box F$) throughout a sequence, if it holds in all suffixes of the sequence. (8) An action holds

for a sequence, iff it holds over the first two states (the first step).

Liveness proofs require reasoning about the ability to carry out an action, that is whether or not an action is *enabled*: an action $\mathcal{A}$ is enabled in a state s if it is possible to do an $\mathcal{A}$ step, meaning there exists a result t state such that $(s, t) \vDash \mathcal{A}$:

$$s \vDash \mathit{Enabled}\ \mathcal{A} \quad \mathit{iff} \quad \exists t \in \mathbf{St}.\ (s, t) \vDash \mathcal{A} \tag{2.4}$$

The definition of flexible quantification requires some auxiliary definitions: firstly, the equivalence of two sequences σ and τ up to a variable x, written $\sigma =_x \tau$: with the exception of x, all variables of all corresponding states in the sequences are equal; secondly, $\natural$ removes all, but infinite sequences of stuttering steps of a sequence:

$$\begin{array}{lcl} \sigma =_x \tau & \triangleq & s_i(v) = t_i(v) \textit{ for all } i \geq 0 \textit{ and for all } v \neq x. \\ \natural\sigma & \triangleq & \textbf{if } s_i = s_0 \textit{ for all } i \geq 0 \\ & & \quad \textbf{then } \langle s_0, s_0, s_0, \cdots \rangle \\ & & \quad \textbf{else if } s_1 = s_0 \textbf{ then } \natural(\sigma|_1) \textbf{ else } \langle s_0 \rangle \circ \natural(\sigma|_1) \end{array}$$

Flexible[2] and rigid quantification is then defined as follows:

$$\begin{array}{l} \sigma \vDash \mathbf{\exists}\, x.\ F \quad \mathit{iff} \quad \exists \rho, \tau \in \mathbf{St}^{\infty}.\ (\natural\sigma = \natural\rho) \wedge (\rho =_x \tau) \wedge (\tau \vDash F) \\ \sigma \vDash \exists\, c.\ F \quad \mathit{iff} \quad \exists\, c \in \mathsf{Val}.\ \sigma \vDash F \end{array}$$

2.3.3 Reasoning within TLA

Figure 2.2 shows the proof system for TLA, as defined in [120]. Here, F, G and H_c are TLA formulas; P,Q and I are temporal predicates; $\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$ and $\mathcal{M}$ are actions; f and g are state functions; e is a constant expression; c is a rigid variable; and x is a flexible variable. The proof system always attempts to reduce temporal formulas into the action level, and is expressed in *Raw TLA* (RTLA) which is a non-stuttering invariant super-set of TLA. This is required since actions are not stuttering invariant. Rules (STL1)-(STL6) are standard temporal logic rules. Rule (TLA1) introduces an induction principle to prove properties of the form $\Box P$, while (TLA2) is used in refinement proofs. Rule (INV1) is used to prove that a program satisfies an invariant I. Such invariant proofs often require a strengthening of the action $\mathcal{N}$ by other (weaker) invariants. This is achieved by rule (INV2).

(WF1) is used to prove leads-to properties $P \rightsquigarrow Q$ $(\Box(P \Rightarrow \Diamond Q))$, which means that it is always the case that if P, then eventually Q. This is proved from a weak fairness

[2]Please see [120] for the reason of the complexity of $\mathbf{\exists}$'s definition.

(STL1) $\dfrac{\vdash F \text{ is tautological}}{\vdash \Box F}$

(STL2) $\vdash \Box F \Rightarrow F$

(STL3) $\vdash \Box\Box F \equiv \Box F$

(STL4) $\dfrac{\vdash F \Rightarrow G}{\vdash \Box F \Rightarrow \Box G}$

(STL5) $\vdash \Box(F \wedge G) \equiv (\Box F) \wedge (\Box G)$

(STL6) $(\Diamond\Box F) \wedge (\Diamond\Box G) \equiv \Diamond\Box(F \wedge G)$

(LATTICE) $\succ$ is a well-founded partial order on a set S

$$\frac{\vdash F \wedge (c \in S) \Rightarrow \big(H_c \rightsquigarrow (G \vee \exists\, d \in S.\ (c \succ d) \wedge H_d)\big)}{\vdash F \Rightarrow \big(\exists\, c \in S.\ H_c) \rightsquigarrow G\big)}$$

(TLA1) $\dfrac{\vdash P \wedge (f' = f) \Rightarrow P'}{\vdash \Box P \equiv P \wedge \Box[P \Rightarrow P']_f}$

(TLA2) $\dfrac{\vdash P \wedge [\mathcal{A}]_f \Rightarrow Q \wedge [\mathcal{B}]_g}{\vdash \Box P \wedge \Box[\mathcal{A}]_f \Rightarrow \Box Q \wedge \Box[\mathcal{B}]_g}$

(INV1) $\dfrac{\vdash I \wedge [\mathcal{N}]_f \Rightarrow I'}{\vdash I \wedge \Box[\mathcal{N}]_f \Rightarrow \Box I}$

(INV2) $\vdash \Box I \Rightarrow (\Box[\mathcal{N}]_f \equiv \Box[\mathcal{N} \wedge I \wedge I']_f)$

(WF1)

$$\frac{\begin{array}{l}\vdash P \wedge [\mathcal{N}]_f \Rightarrow (P' \vee Q') \\ \vdash P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \Rightarrow Q' \\ \vdash P \Rightarrow \mathit{Enabled}\ \langle \mathcal{A} \rangle_f\end{array}}{\vdash \Box[\mathcal{N}]_f \wedge \mathit{WF}_f(\mathcal{A}) \Rightarrow (P \rightsquigarrow Q)}$$

(WF2)

$$\frac{\begin{array}{l}\vdash \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \Rightarrow \langle \overline{\mathcal{M}} \rangle_{\overline{g}} \\ \vdash P \wedge \overline{P'} \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \Rightarrow \mathcal{B} \\ \vdash P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \Rightarrow \mathit{Enabled}\ \langle \mathcal{A} \rangle_g \\ \vdash \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathit{WF}_f(\mathcal{A}) \wedge \Box F \\ \qquad \wedge \Diamond\Box \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \Rightarrow \Diamond\Box P\end{array}}{\vdash \Box[\mathcal{N}]_f \wedge \mathit{WF}_f(\mathcal{A}) \wedge \Box F \Rightarrow \overline{\mathit{WF}_g(\mathcal{M})}}$$

(SF1)

$$\frac{\begin{array}{l}\vdash P \wedge [\mathcal{N}]_f \Rightarrow (P' \vee Q') \\ \vdash P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \Rightarrow Q' \\ \vdash \Box P \wedge \Box[\mathcal{N}]_f \wedge \Box F \\ \qquad \Rightarrow \Diamond \mathit{Enabled}\ \langle \mathcal{A} \rangle_f\end{array}}{\vdash \Box[\mathcal{N}]_f \wedge \mathit{SF}_f(\mathcal{A}) \wedge \Box F \Rightarrow (P \rightsquigarrow Q)}$$

(SF2)

$$\frac{\begin{array}{l}\vdash \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \Rightarrow \langle \overline{\mathcal{M}} \rangle_{\overline{g}} \\ \vdash P \wedge \overline{P'} \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \wedge \Rightarrow \mathcal{B} \\ \vdash P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \Rightarrow \mathit{Enabled}\ \langle \mathcal{A} \rangle_g \\ \vdash \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathit{SF}_f(\mathcal{A}) \wedge \Box F \\ \qquad \wedge \Box\Diamond \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \Rightarrow \Diamond\Box P\end{array}}{\vdash \Box[\mathcal{N}]_f \wedge \mathit{SF}_f(\mathcal{A}) \wedge \Box F \Rightarrow \overline{\mathit{SF}_g(\mathcal{M})}}$$

(E1) $\vdash F[f/x] \Rightarrow \mathbf{\exists} x.\ F$

(E2) $\dfrac{\vdash F \Rightarrow G \quad x \text{ does not occur free in } G}{\vdash (\mathbf{\exists} x.\ F) \Rightarrow G}$

(F1) $\vdash F[e/c] \Rightarrow \exists\, c.\ F$

(F2) $\dfrac{\vdash F \Rightarrow G \quad c \text{ does not occur free in } G}{\vdash (\exists\, c.\ F) \Rightarrow G}$

Figure 2.2: TLA proof rules [120].

assumption, while (WF2) deduces a weak fairness condition from another. (SF1) and (SF2) are the strong fairness versions of (WF1) and (WF2).

The following simple example illustrates the application of rules in a proof. Firstly, a standard inductive definition of Even, the set of even numbers, is given:

$$(\mathsf{even\_base})\ 0 \in \mathsf{Even} \qquad\qquad (\mathsf{even\_step})\ \frac{X \in \mathsf{Even}}{X+2 \in \mathsf{Even}}$$

and the following property is verified:

$$x = 0 \wedge \Box[x' = x + 2]_x \Rightarrow \Box(x \in \mathsf{Even})$$

The proof is given in a backwards fashion, where the goal is manipulated to resemble the assumption: firstly, by (even_base), $x \in \mathsf{Even}$ holds for the initial state; then, rule (INV1) reduces the proof to the action level, requiring the proof that $x' \in \mathsf{Even}$ as a result of the action; next, the "unchanged step", which follows from the subscript, and the "step case" $x' = x + 2$ are split: the "unchanged case" is trivial; while the "step case" is trivial after applying rule (even_step). The proof is shown by the following proof tree, and should be read bottom-up:

$$\dfrac{\dfrac{\dfrac{\dfrac{x \in \mathsf{Even} \Rightarrow x \in \mathsf{Even}}{x \in \mathsf{Even} \wedge x' = x \Rightarrow x' \in \mathsf{Even}}\,(\mathrm{simp}) \quad \dfrac{\dfrac{x \in \mathsf{Even} \Rightarrow x \in \mathsf{Even}}{x \in \mathsf{Even} \Rightarrow x + 2 \in \mathsf{Even}}\,(\mathsf{even\_step})}{x \in \mathsf{Even} \wedge x' = x + 2 \Rightarrow x' \in \mathsf{Even}}\,(\mathrm{simp})}{x \in \mathsf{Even} \wedge [x' = x + 2]_x \Rightarrow x' \in \mathsf{Even}}\,([\cdots]_{\cdots})}{x \in \mathsf{Even} \wedge \Box[x' = x + 2]_x \Rightarrow \Box(x \in \mathsf{Even})}\,(\mathsf{INV1})}{x = 0 \wedge \Box[x' = x + 2]_x \Rightarrow \Box(x \in \mathsf{Even})}\,(\mathsf{even\_base})$$

(F1) and (F2) are the introduction and elimination rules for existential quantification of rigid variables, and follow from standard introduction and elimination rules for $\exists$. Note that e is a constant expression and c is a rigid variable. (E1) is the introduction rule for $\mathbf{\exists}$, while (E2) is the elimination rule. These are comparable to (F1) and (F2). However, f is a *state function* while x is a flexible variable. (E1) and (E2) are used with the internal, or $\mathbf{\exists}$-bound, variables, as shown in (2.2) for example. Let Φ be a high-level specification and Π a refinement of it, without hiding the internal variables $x_1, x_2, \cdots x_n$ and $y_1, y_2, \cdots y_m$, respectively. The refinement proof reduces to showing that

$$\mathbf{\exists}\, y_1, y_2, \cdots y_m.\ \Pi \Rightarrow \mathbf{\exists}\, x_1, x_2, \cdots x_n.\ \Phi. \tag{2.5}$$

In the proof of (2.5),

$$\Pi \Rightarrow \exists\, x_1, x_2, \cdots x_n.\ \Phi \tag{2.6}$$

is normally first proved, and then (E2) is applied to $\exists$-bind $y_1, y_2, \cdots y_m$. The proof of (2.6) is the main part of the proof (2.5). As with any "existential proof", the proof requires obtaining witnesses $f_1, f_2, \cdots, f_n$ for the $\exists$-bound variables $x_1, x_2, \cdots, x_n$, respectively. Each witness f_i is a state-function, expressed by all free-variables in Π, and the sum of all the witnesses $f_1, f_2, \cdots, f_n$ is called the *refinement mapping* [3]. Moreover, $\Phi[f_1/x_1, f_2/x_2, \cdots, f_n/x_n]$ is written $\overline{\Phi}$, and $\overline{\Phi} \Rightarrow P$ means that P holds *under this refinement mapping*. The substitution, and thus the $\overline{\cdots}$ operator, distribute over most operators, and all propositional and safety operators. For example, $\overline{\Box(\cdots \wedge \cdots)} \equiv \Box(\overline{\cdots} \wedge \overline{\cdots})$. However, as illustrated in rules (WF2) and (SF2), it does not distribute over *Enabled*, *WF* and *SF* [120]. Now, to prove (2.6) $\Pi \Rightarrow \overline{\Phi}$, is first proved, and then (F1) is appled to $x_1, x_2, \cdots, x_n$ $(f_1, f_2, \cdots, f_n)$ to show (2.6). For example, the simple refinement

$$(x = 0 \wedge \Box[x' = x + 2]_x) \Rightarrow (\exists\, y.\ y = 1 \wedge \Box[y' = y + 2]_y),$$

is verified by the $[(x+1)/y]$ substitution. Thus $\overline{F}$ becomes $F[(x+1)/y]$. This reduces the goal to the following, which can be verified by (TLA2) and (STL4):

$$(x = 0 \wedge \Box[x' = x + 2]_x) \Rightarrow \big((x+1) = 1 \wedge \Box[(x+1)' = (x+1) + 2]_{(x+1)}\big).$$

Sometimes the refinement mapping requires augmenting the specification by an *auxiliary variables* [3]. For example, a *history* variable, which records past information, can be added to a specification without altering the system behaviour.

2.3.4 TLA*

TLA* [144] is a generalisation of TLA. It is partly a result of a lack of a proper proof-system for TLA, which has caused problems in prevoius mechanisations of TLA [65, 115, 128, 143, 197, 202]. The discussion of these embeddings are delayed to Section 3.1.1. It is motivated by two related TLA shortcomings. Firstly, strong syntactic restrictions so that particularly many specifications and properties cannot be expressed in a natural way. In particular, $\Diamond(F \wedge \Diamond G)$ can be expressed, whilst the action counterpart $\Diamond\langle P \wedge \Diamond\langle G\rangle_v\rangle_w$ is not a well-formed TLA term. Secondly, TLA does not have an

adequate proof system[3]. Lamport [120] only shows a relative completeness result for standard monolithic specifications, and the proof system resorts to the non invariant stuttering RTLA. The proof system becomes particularly important with respect to mechanisation, and Merz argues that TLA* is better suited for mechanical verification than TLA [144].

TLA* replaces the action layer of TLA with a pre-formula layer, subsuming the temporal level, which in TLA* is called the formula layer. The priming operator is omitted, and replaced by a local $\circ F$ operator, where F is a formula. This is only allowed in a pre-formula, and thus deviates from the global 'next' operator. Merz [144] addresses only the propositional case, which is extended here to the predicate case in Chapter 3.

Let $\mathcal{V}$ be the set of all atomic propositions, hence $v \in \mathcal{V}$ is a formula. All propositional operators are also both formulas and pre-formulas, hence if, for example, F and G are formulas then $F \wedge G$ is a formula[4]. Furthermore, if F is a formulas then F and $\circ F$ are pre-formulas and $\Box F$ is a formula. Finally, if $v \in \mathcal{V}$ and P is a pre-formula, then $\Box[P]_v$ is a formula. Sequences, abbreviations etc. are similar to TLA, but note that only the propositional case is addressed, thus a state is a predicate on $\mathcal{V}$. The semantics of TLA* is then defined inductively as follows:

(1) $\sigma \vDash v$ *iff* $s_0(v)$ $(\textit{for } v \in \mathcal{V})$.
(2) $\sigma \vDash \neg A$ *iff* $\sigma \vDash A$ *does not hold.*
(3) $\sigma \vDash A \Rightarrow B$ *iff* $\sigma \vDash A$ *implies* $\sigma \vDash B$.
(4) $\sigma \vDash \Box F$ *iff* $\sigma|_i \vDash F$ *holds for all* $i \geq 0$.
(5) $\sigma \vDash \Box[P]_v$ *iff* $s_i(v) = s_{i+1}(v)$ *or* $\sigma|_i \vDash P$ *holds for all* $i \geq 0$.
(6) $\sigma \vDash \circ F$ *iff* $\sigma|_1 \vDash F$ *holds.*

The semantics is similar to standard possible world semantics for modal logics [28], with the difference that stuttering invariance of TLA* formulas are ensured. (1) Since only the propositional case is discussed, $v \in \mathcal{V}$ is either true or false. Thus, v satisfies a behaviour iff it holds in the first state. (2) A behaviour satisfies $\neg A$ iff it does not satisfy A. (3) A behaviour satisfies $A \Rightarrow B$ iff it satisfies B when it satisfies A. (4) A behaviour σ satisfies $\Box F$ iff F holds in all suffices of σ. (5) A behaviour σ satisfies $\Box[P]_v$ iff for all suffixes $\sigma|_i$ of σ, the first step is either a stuttering step, or P holds for $\sigma|_i$. (6) A behaviour σ satisfies $\circ F$ iff F satisfies the tail $\sigma|_1$ of σ.

Flexible quantification has the same semantics as in TLA and is therefore not

[3] In [2], Abadi defines a proof system for an earlier version of TLA. However, TLA has changed drastically since then.

[4] F and G are used for formulas, P and Q for pre-formulas, while A and B are both.

(ax0)	$\vdash F$ whenever F is tautological	(pax0)	$\mid\!\sim P$ whenever P is tautological
(ax1)	$\vdash \Box F \Rightarrow F$	(pax1)	$\mid\!\sim \circ\neg F \equiv \neg \circ F$
(ax2)	$\vdash \Box F \Rightarrow \Box[\Box F]_v$	(pax2)	$\mid\!\sim \circ(F \Rightarrow G) \Rightarrow (\circ F \Rightarrow \circ G)$
(ax3)	$\vdash \Box[F \Rightarrow \circ F]_F \Rightarrow (F \Rightarrow \Box F)$	(pax3)	$\mid\!\sim \Box F \Rightarrow \circ\Box F$
(ax4)	$\vdash \Box[P \Rightarrow Q]_v \Rightarrow (\Box[P]_v \Rightarrow \Box[Q]_v)$	(pax4)	$\mid\!\sim \Box[P]_v \equiv [P]_v \wedge \circ\Box[P]_v$
(ax5)	$\vdash \Box[\circ v \not\equiv v]_v$	(pax5)	$\mid\!\sim \circ\Box F \Rightarrow \Box[\circ F]_v$
(mp)	$\dfrac{\vdash F \qquad \vdash F \Rightarrow G}{\vdash G}$	(pmp)	$\dfrac{\mid\!\sim P \qquad \mid\!\sim P \Rightarrow Q}{\mid\!\sim Q}$
(sq)	$\dfrac{\mid\!\sim P}{\vdash \Box[P]_v}$	(pre)	$\dfrac{\vdash F}{\mid\!\sim F}$ (nex) $\dfrac{\vdash F}{\mid\!\sim \circ F}$

Figure 2.3: TLA hetereogenous proof system [144].*

discussed it here. Also note that to ensure stuttering invariance F must be a formula in $\circ F$ and $\Box F$. Merz [144] shows by example why this is the case.

Figure 2.3 shows the proof system of TLA*. It uses two provability relations: $\vdash$ for formulas; and $\mid\!\sim$ for pre-formulas. In particular note that: in (ax3) the subscript F is short hand for the free variables of F; (ax5) asserts that pre-formula P in $\Box[P]_v$ is evaluated only when v changes value; (pax1) expresses linear time; (sq) is used to convert a pre-formula into a formula; and (pre) and (nex) convert in the other direction. In the predicate case, the TLA rules from Figure 2.2 can be derived using the rules from Figure 2.3 and standard predicate calculus rules.

Formulas contain the interesting properties. Pre-formulas are merely auxiliary in order to achieve this. Thus, Merz also suggests a *homogenous* proof system, where the $\mid\!\sim$ relation is omitted. It requires some changes to the (pax) axioms, which are then "boxed" by the (sq) rule. Merz argues that this will be better suited for mechanical verification. However in the attempt to mechanise TLA* in Chapter 3, the proof system of Figure 2.3 proved easier to use. Thus, the homogenous variant is not discussed further.

2.3.5 TLA$^+$

TLA is lifted in the sense that it is generic with respect to the underlying data structures. TLA$^+$ [124] is a full specification language[5] which combines TLA with a variant of first-order Zermelo Fraenkel (ZF) set-theory with choice. It provides facilities to define operators and recursive functions. Further, it allows specifications to be divided into parametric hierarchical modules, albeit this is simply syntactic sugaring to simplify specifications. Moreover, it adds a module system to TLA, where modules are

[5]See [145] for a detailed discussion of the underlying logic of TLA$^+$.

imported by the EXTENDS statement, and a set of pre-defined modules are included. For example, the natural numbers module *Naturals* provides Peano-numbers, while *Integers* extends this to integers, and *Reals* provides real numbers. The *Sequence* module formalises tuples, lists and sequences – which are the same since TLA$^+$ is type-free. *Seq*(T) defines a sequence of type T, where by type we mean a set of constants. It also provides operators on sequences, like *Len*, *Head* and *Tail*, which have the obvious meaning – and the binary $\cdot$ which adds a sequences to the end of of the first. Sequences can also be written $\langle \cdots \rangle$, where $\langle\rangle$ denotes an empty sequence. A TLA$^+$ operator is semantically the same as replacing it with its definition, and arguments are given inside brackets $(\cdots)$. Functions, on the other hand, are not purely syntactic, and support recursion. Here arguments are given inside square brackets $[\cdots]$. Note that the valid input "type" must be given in the definition to support the recursion. For example, $f[a \in T] \triangleq \cdots$ defines a function f, where the input a must be a member of the set T. IF and CASE expressions are also supported.

Recently, TLA$^+$ has been extended with a hierarchical proof language [122], and support is added for recursion and lambda notation for operators. This is called TLA^{+2} [126]. TLA$^+$ is a proper sub-language of TLA^{+2} [126], and TLA^{+2} will eventually replace TLA$^+$. All work in this thesis has been in TLA$^+$. However, since TLA$^+$ is a proper sub-language of TLA^{+2} the work in this thesis is just as valid in TLA^{+2}, albeit better mechanisms may exists in TLA^{+2} to achieve a more direct embedding of the Hume semantics.

2.4 Automated reasoning

Automated reasoning techniques attempt to automate the proofs of system properties using computer programs. The verification problem is, in general, undecidable for sufficiently expressive programming languages. Thus, mechanical verification often requires user interaction, and *automated reasoning techniques* attempts to reduce this interaction as much as possible. Broadly speaking, there are then two types of mechanical verification techniques:

- an *algorithmic* or *state-based* approach known as *model checking*, which is discussed in Section 2.4.1;
- a *deductive* or *proof-based* approach known as *theorem proving*, which is discussed in Section 2.4.2, followed by an introduction to the Isabelle theorem prover in Section 2.4.3.

Section 2.4.4 discusses various mechanical verification tools for formal software verification.

2.4.1 Model checking

Model checking [43] was independently developed by Clarke and Emerson [42] and by Queille and Sifakis [168]. Clarke, Emerson and Sifakis received the 2008 Turing award for this work. In the model checking problem, the model M of the system is represented as a Kripke structure over a quadruple $M = (S, S_0, R, L)$:

1. S is a finite set of states.
2. $S_o \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a total transition for the states.
4. $L : S \rightarrow 2^{AP}$ is a labelling function that labels each state $s \in S$ with the set of atomic propositions that hold in state s.

If liveness properties are used then the Kripke structure is extended with a fairness constraint $F \subseteq 2^S$. The *model checking problem* for a temporal property π is then expressed as $M \vDash \pi$, meaning 'given the model M, does π hold'. This is verified by traversing the Kripke structure M for violations of π. If a violation is found a *counter-example* is given. The advantage of model checking is full automation and counter-example generation, which can be helpful in identifying and repairing errors. However, it requires the model to be finite and is not as expressive as theorem provers. Moreover, it does not say *why* a property holds: often a property holds for other reasons than those assumed. The largest problem with model checkers is the *state space explosion problem*: a computer only has a finite sized memory, and at some point the program becomes too big, and the model checker will fail.

With regard to state representation, *explicit state* model checking is the simplest form and represents the complete state. TLC [124, 203] is an explicit time model checker. In *automata* based model checkers, automata are used to represent both specification and property, and validation is by containment of the property in the automaton. Spin [97] is an example of this type of model checker. *Symbolic* model checking uses the symbolic OBDD representation of Kripke structures, while fixed points [140] are used to verify properties. SMV [140] is an example of a symbolic model checker.

There are also various techniques to reduce both the search time and space required for a given model checking problem. These techniques may rely on the state representation. For example, Spin is an *on-the-fly* model checker, where the state space is created

during search. Thus, some errors can be found early, without creating the full state space, which enables identification of some errors even if the full state space cannot be generated. Spin also supports *partial-order reduction* which explores commutativity of concurrent transactions to reduce the state space. There are also techniques that explore *symmetry* in the search tree of the state space, and *abstraction* techniques that attempt to simplify the model to reduce the steps. For example, *predicate abstraction* [52] combines model checking with theorem proving to verify that the abstraction is indeed correct. Predicate abstraction and *bounded model checking*, which uses a fixed number n of steps of M, and then check whether a property violation can occur in n or fewer steps, can be used to model check infinite state systems.

As a concluding remark, model checking is suitable for finite systems, which can be expressed by a finite state machine (FSM), and is well-suited for *control-centric* problems which can often be expressed as an FSM.

TLA$^+$ supports model checking via the TLC tool [124, 203], where the specification must import the *TLC* module. Only a decidable finite subset of the language is supported, whilst all constants must be given a value. This is achieved by augmenting the model checker with a configuration file, restricting sets into acceptable subsets, and giving constants values. TLC version 2.0 has been used here.

2.4.2 Theorem proving

Theorem proving has a longer history than model checking: the earliest well known system is the Logic Theory Machine (1963) [155]. In a theorem prover deductive techniques are used to verify conjectures. There is a difference between:

- *proof checkers* which, given a conjecture and a proof, check that the proof is correct;
- *automatic theorem provers* which, given a conjecture, attempt to automatically prove it;
- *interactive theorem provers* which, given a conjecture, attempt to verify it by interaction with a user.

In the area of automatic theorem provers, Robinson [172] has carried out seminal work on resolution. Automatic theorem provers are undecidable for sufficiently expressive logics, like first-order, or higher-order, predicate logic. However, for the first-order predicate calculus, it is *semi-decidable*: if a conjecture holds it will be found, but the prover may never terminate.

An *interactive theorem prover*, also called a *proof assistant*, is something between a proof checker and an automatic theorem prover, depending on the level of automation supported. At one end of the spectrum, the user guides every small step in the deduction, while at the other end the user only gives high-level guidance of the proof. Here, the smaller steps are handled by special made *tactics* and *decision procedures*: a tactic is a method/program that performs one or more steps in a proof; while a decision procedure is an algorithm that attempts to solve an (often undecidable) decision problem.

Another classification of theorem provers is based on their underlying logic: *classical provers* are based on higher-order logic, for example Isabelle/HOL [158], HOL [80] and PVS [164]; *constructive provers*, like Coq [25] and Agda [50], are based on constructive logics; *set-theoretic provers*, like Larch [75] and Isabelle/ZF [158], are based on some sort of set theory; while *first-order provers*, like ACL2 [117] and Isabelle/FOL [158] uses first-order logic.

2.4.3 The Isabelle/HOL theorem prover

> *"Don't write a theorem prover. Try to use someone else's."*
>
> – Lawrence C. Paulson [165]

Isabelle is a framework for interactive theorem proving, following the *LCF (Logic for Computable Functions)-approach* [82], where ML [151] is used as an implementation language, and abstract data type constructors are used to ensure secure inferences. It contains a generic meta-logic, called *Isabelle/Pure*, which is a minimal version of higher order logic, used to define and manipulate object logics. Initially, ML was used to both specify and verify systems and conjectures. Now, this is handled by a framework which allows much more sophisticated specifications and infrastructure called Isar [200]. Here, *Isabelle/HOL* [158] an implementation of higher-order logic, is assumed. This is by far the most developed object logic in Isabelle. Note that to be precise Isabelle/Isar/HOL should have been written, however, Isar is now used in all Isabelle object logics and is thus omitted.

Isabelle/HOL follows the *definitional approach*, where new axioms should not be asserted. Instead, constants, types and functions should be defined, and properties about these should be proven. Here, abstract syntax can be defined using mixfix syntax and mathematical symbols, which is then translated into the concrete syntax. To enable the definitional approach, Isabelle/HOL allows definitions using for example inductive datatypes, inductively defined sets and several ways of writing recursive functions.

```
lemma A ∧ B ⟶ B ∧ A
  apply (rule impI)
  apply (erule conjE)
  apply (rule conjI)
  apply assumption
  apply assumption
  done
```

```
lemma A ∧ B ⟶ B ∧ A
proof (rule impI)
  assume A ∧ B
  then obtain A and B ..
  then show B ∧ A ..
qed
```

Figure 2.4: Example of a procedural (left) and a structural (right) proof in Isabelle/HOL.

Isabelle formalisations are structured into *theories*, and the theories are organised as graphs. Further, modularity is supported by *axiomatic type classes* [166, 199] and *locales* [20, 21, 116]: axiomatic type classes allow a class of types to be defined in terms of their properties, and is comparable to Haskell's type classes [196]; Isabelle is built up around theory contexts, and locales introduce local contexts.

Isabelle/Pure contains a quantifier $\bigwedge$, to express arbitrary, but fixed entities; $\Longrightarrow$ expresses logical entailment; and $\equiv$ for equality. Here, $A \Longrightarrow B \Longrightarrow C$, $[\![A; B]\!] \Longrightarrow C$ and '**assumes** A **and** B **shows** C' are semantically equivalent. Rules are represented in Gentzen style natural deduction [76]. Care must be taken in separating the meta-level connectives with object-level connectives, which here refer to Isabelle/HOL. With regard to types; the product type is represented as $\times$ and function types are written $\Rightarrow$. Note that object level implication is written $\longrightarrow$, while $=$ is used for both equality and equivalence. Moreover, **lemma** and **theorem** are both used to express a theorem in Isabelle.

Isar allows the user to work with a sophisticated interface compared to the ML level. However, ML is still used to implement higher level *tactics*, defined using primitive inference rules, previously defined rules and tactics, and previously proved theorems. A tactic can then be lifted into an Isar *method*. In the original ML representation, proofs were formalised in a backwards fashion, where the goal was reduced to sub-goals by resolution, directed by a sequence of ML commands which specified the tactics to apply. This representation is called a *procedural* proof. This proof style is still supported in ML, albeit Isar methods and not tactics are applied. In addition, Isar also supports forward *structural* proof scripts [157, 200]. Figure 2.4 illustrates a procedural (left) and structural (right) proof of the $A \wedge B \longrightarrow B \wedge A$ conjecture.

The rule method performs introduction resolution, while the erule method performs elimination resolution. The former is normally applied to the goal (right of the rightmost $\Longrightarrow$) while the latter is normally applied to the assumptions (left of the rightmost $\Longrightarrow$). In the procedural example (Figure 2.4, left), the impl rule applied to

the rule method reduces object-level implication $\longrightarrow$ into meta-level entailment $\Longrightarrow$. This reduces the conjecture to $A \wedge B \Longrightarrow B \wedge A$. conjE applied to the *erule* method performs conjecture elimination in the assumption, thus reducing the conjecture to $[\![A; B]\!] \Longrightarrow B \wedge A$. Conjunction introduction (rule conjE) reduces this to the two conjectures $[\![A; B]\!] \Longrightarrow B$ and $[\![A; B]\!] \Longrightarrow A$ which are both proved by the assumption method. The proof is completed by **done**.

A structural proof is encapsulated within **proof** $\cdots$ and **qed**. Here, $\cdots$ can be a list of method applications in a backward style. In the structured proof example (Figure 2.4, right), rule impl is first applied backwards creating the conjecture $A \wedge B \Longrightarrow B \wedge A$. In the proof, the assumption is made explicit by **assume**. From this, the goal $B \wedge A$ must be showed. The .. method attempts to solve a goal by standard introduction rules followed by applying the (given) assumptions. This is used to obtain A and B, from the $A \wedge B$ assumption. A and B are then used to show the goal using the same method. Note that procedural and structural proofs can be interleaved.

A practical verification task will use higher level tactics and tools, instead of the one-step resolution used in Figure 2.4, and Isabelle/HOL provides a number of such tools. Most importantly, is the built in *simplifier*, which performs higher order conditional rewriting using previously proven theorems. The set of used theorems, henceforth the *simplification set*, can be customised by the user. Other important tactics are the *classical reasoner* (the blast tactic), and the *automatic tactic* (auto) which combines the simplifier and classical reasoner.

A Isabelle proof state consists of a list of sub-goals that needs to verified. Isar methods are normally implicitly applied to either the first sub-goal or all the sub-goals. For example, simp applies the simplifier to the first sub-goal, while simp_all and auto applies the simplifier and automatic tactic to all sub-goal. ML level tactics, which are not lifted to the Isar level, can be called by the tactic method, e.g. **apply** (tactic "my_tac"). The ML level is often more flexible, allowing tactics or inference rules to be applied to particular sub-goals.

Proof General for Emacs [16] is the standard user-interface with Isabelle, but it is also possible to execute Isabelle *theories* in a ML shell level. Finally, Isabelle/HOL 2007 has been used here, unless otherwise specified.

2.4.4 Mechanical software verification

Below, examples are given of automated reasoning techniques applied to software verification, with a focus on the source code level.

Verification by model checking

Intel has applied TLC to verify multiprocessor memory designs formalised in TLA^+ [24]. A famous case-study is the Java Pathfinder [193]. Here, a tool translated Java code into the Spin model checker, and found bugs in NASA's Mars Rover [169]. Later versions of the Java Pathfinder abandoned Spin, and instead used a tailor-made model checker for Java programs, called Bandera [51]. There are also purpose built model checkers for many other languages, like Ada [62], Erlang [160] and C [19, 48, 49]. BOGOR [171] is a generic model checker which can be configured to work in a particular way for a particular language. Recently, Microsoft's Terminator tool [48, 49] has drawn most attention. It is used to verify termination of C program, and has later been extended to other liveness properties [47].

Verification by theorem proving

The *Verisoft* project [190] uses Isabelle/HOL to formalise a compiler for a C-like language [132], while [177] formalised a C-like language with dynamic memory allocation, and derived Hoare-triples from it. The *Why/Krakatoa/Caduceus* platform [68] combines three tools. The *Why* tool [66] reads inputs in a specific language dedicated to software verification. It then converts them to verification conditions, and several interactive and automated theorem provers can be applied. *Krakatoa* [138] is a front end for Java programs annotated by *JML* – the Java Modelling Language [131]. Java programs with annotations are translated into the Why input language. *Caduceus* [67] is a front-end for *C* programs annotated with a specialised version of JML. These are also translated into the Why input language. Finally, SPARK [23], an Ada subset with a purpose build theorem prover, has been used for many industrial safety-critical systems.

Combining techniques

Microsoft's Slam [19] uses predicate abstraction to verify drivers written in unannotated C-code. The invariants are automatically inferred from the source code. The Stanford Temporal Prover (STeP) [27] combines automatic- and interactive theorem proving with model checking, and has been developed at Stanford University (USA) since 1994 to verify both temporal safety and liveness properties.

2.5 Programming Languages

In his book on safety-critical systems [180], Storey lists six requirements for programming languages in such domains. They are: *low complexity of definitions* (i.e. high-levelness); *expressive power*; *bounded space and time usage*; *logical soundness*; *security*; and *verifiability*.

Low-level languages depend on the underlying hardware, and include *first-generation* languages (machine-code) and *second-generation* (assembly code) languages. *High-level* languages abstract over the hardware, and may be portable over many platforms. In general, low-level languages tend to be more efficient, while high-level languages are simpler to work with. A high-level language may be *imperative* or *declarative*. Imperative languages, like C, C++, Java and Ada, are based on the theory of *Turing machines* [185]. They describe "how" to compute, represented as a sequence of statements that change the state of a program. *Declarative* languages are more abstract and describe "what" to compute. There are two types of declarative languages: *logical* languages, like Prolog [45], use mathematical logic for computation; and *functional* languages, like ML [151] and Haskell [111], represent computations as applications of mathematical functions.

Functional languages are based on the *lambda calculus* [22], and have many desirable properties: the abstraction over the state implies no mutable objects, hence there are no side effects in a computation. A function can therefore ignore global properties, thus enabling *local reasoning*. Most functional languages are *higher order*, meaning that functions can accept functions as inputs and return functions as results. This enables the principle of *currying* where inputs can be partially applied. The *Church-Rosser* property [41] holds, meaning a result is independent of evaluation order. Functional languages have mainly been successful in academia, and most theorem provers are implemented by a functional language. A problem with *pure* functional languages, as described above, is that some required constructs cannot be handled. For example, I/O has side-effects, hence languages that require I/O must have some *impure* fragments as well, where the mathematical properties described above do not hold. In Haskell this is handled by monads [195], while ML allows programs to contain variables (state).

No programming language fulfils all of Storey's requirement, and many of the requirement are contradicting. For example, for Turing-complete languages space and time bounds are undecidable. However, these are decidable for a (Turing incomplete) *finite state automaton* (FSA) language. A FSA is an abstract model of a computer with a very primitive internal memory, and consists of: a finite number of states; and, transitions between the states based on input and/or the current state. Notwithstanding, they are low level, and working with them is known to be both tedious and time

consuming, and it is easy to introduce errors.

Logical soundness refers to the existence of a sound unambiguous definition of the language, i.e. formal semantics. There are three types of formal semantics. In *operational* semantics the execution of a program is described, which can be seen as an interpreter of a program. Hoare-triples forms an *axiomatic* semantics. While, in *denotational* semantics, each program construct is given a meaning (denotation), which is translated into another language.

Security denotes whether violations of the language definitions are detected before execution, while verifiability denotes whether it can be proved that the program is consistent with its specification, and is the topic of this thesis. Moreover, there are also other requirements to programming languages, for example, UK MoD Standard 00-55 [152] bans assembly code in all MoD applications, and [180] further lists *tool support* and *language expertise* as key factors when choosing a language. Thus, defining a safe subset of an existing language is a common solution, since existing tools like compilers can be applied directly, and users are familiar with the language. Examples of this approach are SPARK [23] for Ada, Misra-C for C and Spade-Pascal for Pascal.

2.6 The Hume programming language

Hume[6] [92] is a high-level expressive programming language which targets resource bounded safety-critical systems. Hume contains a Turing-complete functional language. It targets resource bounded systems, and since time and space properties are undecidable for Turing-complete languages, a *finite state automaton* (FSA) language is built on top. Here, the I/O is also handled. It is normally attempted to work as much as possible in the functional language, known as the *expression layer*, only resorting to the FSA language, known as the *coordination layer*, when required. Since Hume focuses on time and space properties, which are operational requirements, it has been given a formal operational semantics [112].

The Hume coordination layer consists of *boxes* and *wires*, where transitions are achieved in the expression layer within a box, by a set of of *matches* of the form

pattern `->` *expression*.

Boxes can only communicate over wires, and each box has a set of input wires and output wires. Each wire is directional and single-buffered: only one box can read from it and only one box can write to it, although the source and destination may be the

[6]Please see [98] and [64] for details.

```
type Nat = word 64;

box even
 in  (i,st::Nat)
 out (o,st'::Nat)
match
 (n,_) -> (n+1,n)
|(*,n) -> (*,n);

box odd
 in  (i,st::Nat)
 out (o,st'::Nat)
match
 (n,_) -> (n+1,n)
|(*,n) -> (*,n);

wire even(even.st' initially 0, odd.o initially 0) (even.st, odd.i);
wire odd (odd.st' initially 1, even.o)(odd.st, even.i);
```

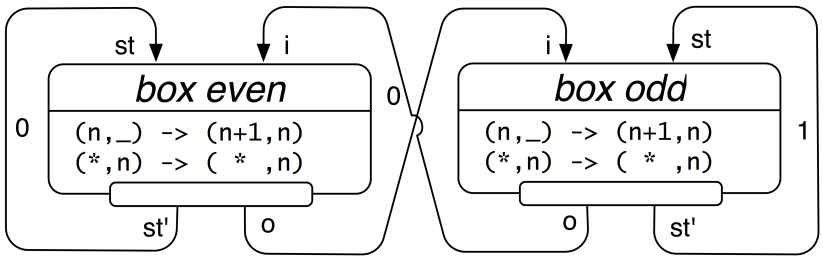


Figure 2.5: Source code and box diagram of a simple Hume example

same box. During execution, the *pattern* of a match is matched against the inputs, and if it succeeds the purely functional *expression* generates output. If it fails the next match is tested until there are no matches left. In that case, nothing happens.

Figure 2.5 shows an example of a simple Hume program. It consists of two connected boxes: `even` and `odd`. In the box diagram of the figure, a wire is represented by an arc, and the beginning and end may have a variable name, which indicates where the wire plugs into the box. A wire can optionally have an initial value as well, shown towards the center of the wire. The program has four wires: both boxes have a feedback wire, in addition to one wire in each direction between the two boxes. Since the expression layer is pure, box state can only be achieved by such feedback wiring. The body of each box is identical: each box has two matches, and the keyword `match` enforces an *unfair* top-down ordering of the match. There is also support for a *fair* least-recently-used ordering of the matches, but this feature has not been used throughout this thesis.

In a *pattern*, '_' succeeds if an input value is present, while '*' ignores the input, i.e. always succeeds. In the event of a match, all inputs, except where the pattern is '*', are consumed from the input wires. These two patterns are combined by '_*', which always succeeds, and consumes the input if present. A variable in the pattern has the same effect as '_', but will bind the variable to the input. Finally, if a pattern is a constant, then the input must have this value. In an *expression*, '*' denotes that no value is produced for that particular output.

Shared between the coordination and expression layer is a rich type system, allowing, for example, inductive data types. Function types are supported in the expression layer, but a wire cannot contain such types. Declared constants may also be used in both layers. Although the expression and coordination layers have well-defined and disjointed tasks, the interface between them is less clear. As will become evident throughout this thesis, this follows from a strong dependency between the layers: firstly, pattern matching is a feature found in many functional languages, and used for example in Hume `case`-expressions. The pattern matching of a "box match" is also used to bind variables, used in the expression, to the input values. On the other side, this pattern matching determines if a box is executed, and which input values are consumed. Thus, pattern matching can be seen as a feature of both layers. Secondly, due to the single-buffering of wires, the result of a transition may have an impact on the scheduling, since a box cannot execute before all the outputs are asserted.

Box scheduling

Hume programs never terminate. Consequently, the scheduler enforces a cyclical execution of the boxes of a program. Previous versions of Hume used a round-robin scheduling algorithm for boxes. Here, each box is executed in a round-robin fashion, and for each execution, the input wires are consumed and the result updated on the output wires. However, this scheduling makes proper analysis of the coordination layer difficult, since the result depends on the order of box execution. Note that the Hume interpreter and compiler implement the scheduling described below. However, the tools implementing the cost models for time and space [29, 90], focuses on the expression layer, and uses a round-robin scheduling of boxes. This is also shown in the operational semantics [112] which the cost model is built on.

Since wires are directional and one-to-one relations, only one box can consume a wire and only one box can write to it. Thus, if all boxes first consume the wires, and then all boxes assert and write to them in a second phase, then the result is independent of how the boxes are executed. In Hume, this is achieved by creating two phases for each execution cycle: in the first phase, each box is run once and attempts to consume inputs and generate outputs; in the second phase, the output changes are resolved on the output wires in a unitary super-step. At the end of each phase a box will be in one of the following states:

- *Runnable*: the box has successfully consumed inputs and asserted outputs.
- *Blocked*: the box has successfully consumed inputs but failed to assert outputs. It will attempt to assert outputs on subsequent cycles.

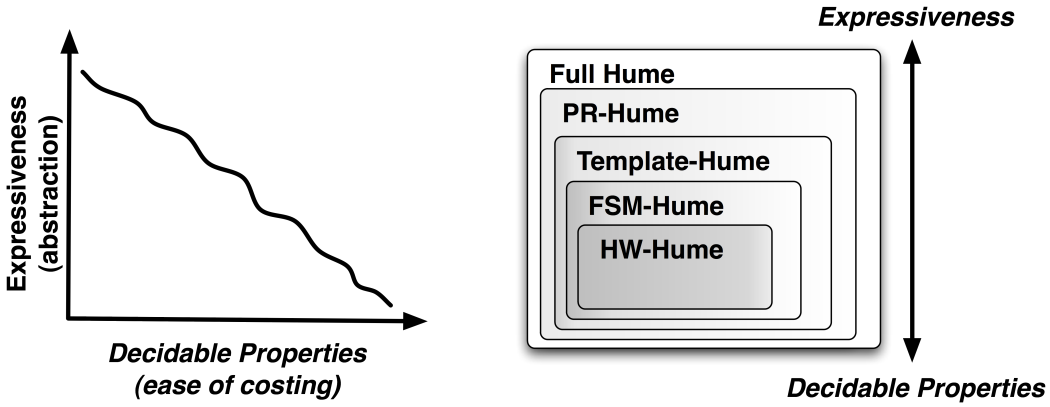


Figure 2.6: Hume levels: decidibility vs. expressiveness

- *Matchfail*: the box has failed to match inputs.

The *lock-step* scheduling then works as follows:

```
for ever
    for each Runnable and Matchfail box
        execute (box)
    super-step
```

The first phase is called the *execute* phase. Here, boxes are matches with the inputs. If it succeeds, the wires are consumed and output is produced. In the unitary *super-step* phase, the result is asserted with the output wires.

Hume levels

Hume attempts to be both high-level and expressive, whilst being able to guarantee bounds on time and space usage statically. This is achieved by introducing a set of *levels*, or sub-languages: each level adds expressive power compared to the level below, while reducing expressiveness compared to the level above; in contrast, the set of decidable properties (i.e. ease of costing) is increased compared to the level above, but reduced compared to the level below. The levels and a graph illustrating this relationship are shown in Figure 2.6:

- *HW-Hume*, or Hardware Hume, is the lowest level. Its low levelness enables a direct embedding of hardware components, no functions are supported and only bits and tuples are allowed. HW-Hume guarantees exact time and space costing and decidable equivalence and termination;

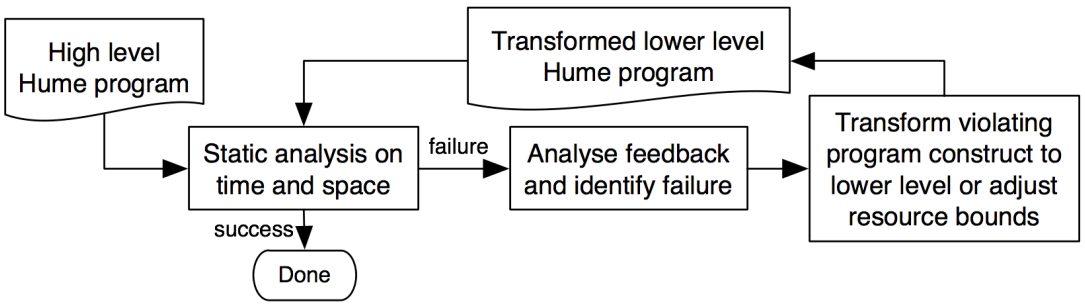


Figure 2.7: Hume methodology

- *FSM-Hume*, or finite state machine Hume, supports non-recursive first-order functions. It allows a richer set of types, but they are limited to finite non-recursive data structures. FSM-Hume guarantees strong bounds on time and space and decidable equivalence and termination;
- *Template-Hume* supports predefined higher-order functions (like `map`) and inductive data structures. Weak bounds on time and space and decidable termination are guaranteed;
- *PR-Hume*, or primitive recursive Hume, extends Template-Hume with primitive recursive functions, and termination is decidable;
- *Full Hume* extends PR-Hume with full recursion, and is undecidable.

Hume methodology

Since resource costing [29, 90], whilst being high-level, is a panacea in Hume, this is reflected in the methodology. Thus, if a high-level program cannot be costed, then either the bounds have to be adjusted, or the program has to be turned into a lower-level version. Since the undecidable constructs are found in the expression layer, a high-to-low level *transformation* involves moving computation into the coordination layer, and costing is reapplied. Figure 2.7 illustrates this methodology.

Hume syntax

Figure 2.8 shows the Hume abstract syntax in BNF form. It incorporates all of HW-Hume. It also allows enumeration types (using `data`), and a richer set of basic types from FSM-Hume. Finally, in some of the examples, non-recursive and primitive recursive functions, and the built in list type is allowed. Thus, a small part of Template-Hume and PR-Hume are allowed. This is illustrated by the shaded area of the levels in

program	::=	$decl_1$; ... ; $decl_n$	$n \geq 1$
decl	::=	*box* | *wire* | *stream* | id pat = *expr*	
	|	`data` id = ⟨ con_1 | ... | con_n⟩	$n \geq 1$
box	::=	`box` id *ins outs* `match` ⟨ $bmatch_1$ | ⋯ | $bmatch_n$ ⟩	$n \geq 1$
ins/outs	::=	⟨ id_1 ::typ_1, ..., id_n ::typ_n ⟩	$n \geq 1$
bmatch	::=	⟨ $bpat_1$, ... , $bpat_n$ ⟩ `->` *bexpr*	$n \geq 1$
match	::=	⟨ pat_1 , ... , pat_n ⟩ `->` *expr*	$n \geq 1$
bexpr	::=	int | bool | word | con | `*`	
	|	var $expr_0$ ⋯ $expr_n$	$n \geq 0$
	|	id $expr_0$ ⋯ $expr_n$	$n \geq 0$
	|	`(` $expr_1$, ... , $expr_n$ `)`	$n \geq 1$
	|	`if` $expr_1$ `then` $expr_2$ `else` $expr_3$	
expr	::=	*bexpr* | `case` *expr* `of` ⟨ $match_1$ | ⋯ | $match_n$ ⟩	$n \geq 1$
typ	::=	`int` int | `word` int | `bool` | *id*	
bpat	::=	*pat* | `*` | `_*`	
pat	::=	*vpat* | `_` | var	
vpat	::=	int | word | bool | con | `(` pat_1 , ... , pat_n `)`	$n \geq 2$
stream	::=	`stream` id `to`/`from` string	
wire	::=	`wire` id `(` $link_1$, ... , $link_n$ `)` `(` $link_1$, ... , $link_m$ `)`	$n, m \geq 1$
link	::=	id_1.id_2 [`initially` *vpat*] | id	

Figure 2.8: TLA-Hume abstract syntax

Figure 2.6. The restrictions have followed directly from features required by the case-studies using TLA, thus the name *TLA-Hume*. Most of the examples are a HW-Hume version, with added FSM-Hume types. However, in a few case-studies, parts of PR-Hume were required. `case` expressions are not supported inside a box, while inductive data types (`data`) are restricted to working as an enumeration type. For simplicity, the syntax does not capture macros (like the `type` abbreviation) and built-in list, boolean and arithmetic operators.

Hume tool support

There is a Hume *interpreter* and *compiler*. The compiler turns Hume into an intermediate Hume Abstract Machine (HAM) representation, which is comparable to the standard abstract machine, like the Java Virtual Machine, with additional extensions for concurrency. HAM is used by the cost models. The code is then compiled into C, which can be compiled to machine code by e.g. `gcc`. The cost model for time is discussed in for example [29], while time and space are treated in for example [90]. Hume attempts to create verified certificates of time and space usage: the `art3` program,

which implements the cost model, returns the actual time/space numbers, and this is used to create a certificate using an embedding of the expression layer in Isabelle/HOL.

2.7 Summary & discussion

This chapter has introduced the key concepts of this thesis: the target language Hume; the automatic reasoning techniques model checking and theorem proving; and the logic TLA to achieve the verification of Hume programs using these reasoning techniques.

TLA fits well into both Hume's design and the required properties, and the motivations behind using TLA is detailed further in Chapter 4. TLA has model checking support via TLC. However, the work in this thesis is seen as first step towards a large verification environment for Hume programs. In parallel to this work, the expression layer has been mechanised in the Isabelle/HOL theorem prover [135], and in such verification environment, integration is highly desirable. A small integration experiment is shown in Section 9.2. Moreover, the focus here is mainly the lower Hume levels. An embedding of the higher data-centric levels, will require theorem proving support. Thus, the focus here is on theorem proving, and the next chapter shows a mechanisation of the TLA* extension of TLA in Isabelle/HOL.

Chapter 3

Mechanising TLA in Isabelle/HOL

"Assumption differs from derivation as theft differs from honest toil."
– Bertrand Russell

3.1 Introduction

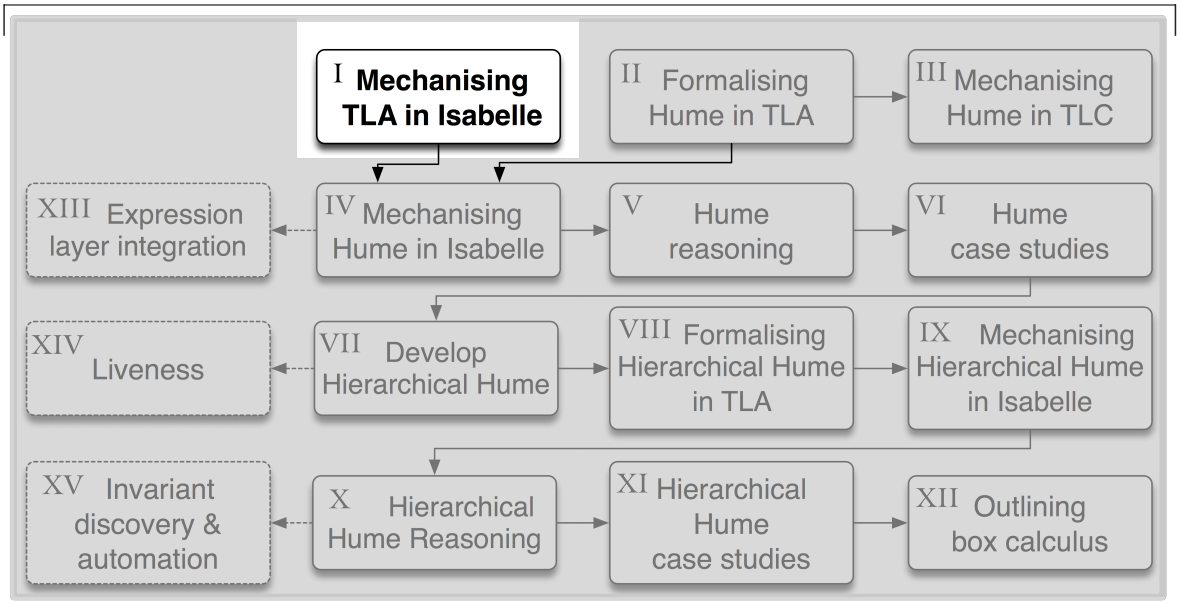


Figure 3.1: Thesis roadmap: Chapter 3

Figure 3.1 highlights which part of the roadmap this chapter implements. This is required to achieve theorem proving capabilities for verifying Hume properties within TLA. However, TLA has previously been mechanised in theorem provers in [65, 115, 128, 143, 197, 202], and Merz's mechanisation [143] is in an older version of Isabelle/HOL. Part of the mechanisation presented here is based directly on [143], and [143] is acknowledged accordingly. However, Merz's work is extended in the following ways: by

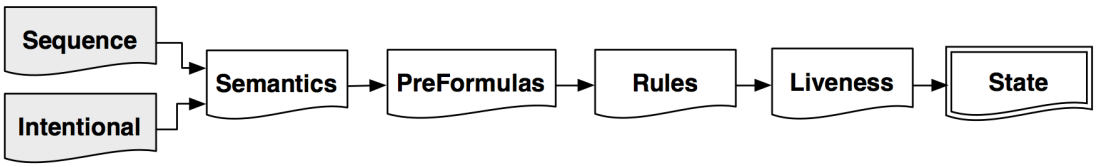


Figure 3.2: The Theory Structure of the TLA Embedding: shaded boxes are independent of the logic; theories independent of the underlying state-space are encapsulated by singe-lined boxes; while double lined boxes are state and logic dependent*

mechanising an independent theory of sequences, enabling possible-world semantics mechanisation and stuttering invariant proofs; TLA* [144], which was developed as a result of [143], is embedded; the TLA* semantics is derived using the sequence theory, and the proof system is derived from semantics, whilst [143] axiomatised both; and finally, the theory is lifted to the Isar level, instead of the old ML-based level.

The main contributions of the work in this chapter are the mechanisation of sequences and the corresponding TLA* semantic and proof system derivations, and the mechanisation of stuttering invariant proof of the TLA* operators. In Merz's TLA* paper [144], he argues that a *homogeneous* proof system is more suitable for mechanisation than a *heterogeneous* version, however, here the opposite was found to be true. Finally, some negative results are provided for the "variable to value function" representation of the state space for modal logics with quantification. This representation was argued for by Ehmety and Paulson [63].

Figure 3.2 shows the theory structure of the mechanisation. The following seven sections discuss each theory separately. The mechanised theorems and lemmas are listed in Appendix A.1. Note that arcs illustrates theory dependencies, and Intentional and Sequences import the Main theory, which is the standard Isabelle/HOL prelude. Further, Intentional and State are more or less, a direct porting of corresponding theories in [143] into Isar. Intentional and Sequence are independent of the mechanised modal logic, while Semantics, Rules and Liveness are independent of the state space representation.

3.1.1 Relevant work

The work presented here is based on Merz's previous work in Isabelle/HOL [143], and extended as described above. Kalvala [115] mechanises TLA in Isabelle/ZF, which she argues is closer to Lamport's definition than Isabelle/HOL [115]. She only supports natural numbers, and sequences are lists of natural numbers, which makes it hard to use for practical verification tasks. Moreover, even though sequences are mechanised,

the proof system is axiomatised and not derived.

[128, 202] mechanises TLA in the HOL system. Tuple representation is used for the state space, and actions and formulas are separated, thus the user must deal with three different types of logical connectives, which is avoided here. [128] discusses quantification over state predicates, achieved by dividing the state space into the bound and unbound variables. It is hard to understand how and why this is correct, how it relates to Lamport's original definition, and whether or not stuttering invariance is preserved. Furthermore, liveness properties are not discussed.

Wahab's mechanises TLA in PVS [197]. He does not attempt to separate actions from formulas, and stuttering invariance, the proof system and actual verification is not discussed. Thus, it is unclear if this system actually works. The mechanisation of $\natural$ in the Sequence theory follows from this work.

TLP [65] is the first known mechanisation of TLA, albeit it is now abandoned. It works by translating TLA specifications into LP proof scripts, which are then verified by the first-order LP prover. Action and temporal proofs are there treated separately. TLA is also used in VSE [17, 173], although the details of this work is not known. Finally, none of the above work mechanises the TLA* logic.

3.2 The Intentional theory

In higher-order logic, every proof rule has a corresponding tautology. Thus, if $F \vdash G$ then $\vdash F \Rightarrow G$, and this is known as the *deduction theorem* [94, 182]. Isabelle implements this since object-level implication ($\longrightarrow$) and meta-level entailment ($\Longrightarrow$) are not semantically distinguished. However, the deduction theorem does not hold for most modal and temporal logics [124, page 95][143]. For example $A \vdash \Box A$ holds, meaning that if A holds in any world, then it always holds. However, $\vdash A \Rightarrow \Box A$, stating that A always holds if it initially holds, is not valid.

Merz [143] overcame this problem by creating an Intentional logic. It exploits Isabelle's axiomatic type class feature [199] by creating a type class world, which provides Skolem constants to associate formulas with the world they hold in. The class is trivial, not requiring any axioms. Here, formulas are of the type

```
types  ('w,'a) expr  = 'w ⇒ 'a
       'w form       = ('w,bool) expr
```

where type variable 'w belongs to the world type class, and written as w ⊨ F instead of F w for a formula F of type 'w form. Validity of a formula is defined as follows:

```
Valid   :: 'w form ⇒ bool (⊢ _)
⊢ F     ≡   ∀ w. w ⊨ F
```

In both ⊢ F and w ⊨ F, F must be *lifted* into the world class. Postfixing the constant definition with (⊢ _) in the definition of Valid abuses the Isabelle syntax translation feature. In the Isabelle theory it is necessary to explicitly introduce a non-terminal lift, and introduce the ⊢ explicitly using this non-terminal. However, to simplify the reading these details are ignored. To lift a formula F, lifting functions are defined for the required Isabelle/HOL types. Firstly, lifted constants, or rigid variables, are prefixed by # and are independent of the world they are in:

```
consts    :: 'a ⇒ ('w,'a) expr (#_)
w ⊨ #c    ≡   c .
```

Secondly, since Isabelle functions are pure, they are lifted by lifting their parameters. A lifted function receives the curried parameters separated by a comma, encapsulated by <···>. Functions are explicitly lifted based on the number of parameters, e.g.

```
lift2 :: ('a ⇒ 'b ⇒ 'c) ⇒ ('w,'a) expr ⇒ ('w,'b) expr ⇒ ('w,'c) expr   (_<_,_>)
w ⊨ f<x,y> ≡   f (w ⊨ x) (w ⊨ y)
```

lifts a two parameter function. Similar function are defined for up to four parameters. Standard HOL notation is reused and explicitly lifted into the world type to ease specification. This includes: pairs; finite sets and set operators; lists and list constructors; boolean connectives; (rigid) quantifiers; arithmetic operators; and if _ then _ else. ⊨ distributes over these operators, e.g. w ⊨ (A ∧ B) is equal to (w ⊨ A) ∧ (w ⊨ B), and w ⊨ (∀ v. P v) equals ∀ v. (w ⊨ P v) which equals ∀ v. P (w ⊨ v). Finally, in both ⊢ and ⊨, the formula is implicitly lifted. For domains outside these operators, a formula is implicitly lifted by the LIFT combinator, where LIFT F ≡ λ s. F s.

3.3 The Sequence theory

Many possible world based logics require that sequences are *possibly infinite*, i.e. they must be able to deal with both finite and infinite sequences. Devillers et al [53] discusses several variants of this kind of sequence which are implemented in HOL, Isabelle and PVS. One approach represents a sequence as a nat ⇒ 'a option function where **datatype** 'a option = None | Some 'a and None identifies the end of a finite sequence. Another approach uses a union type where finite sequences are of type 'a list and infinite sequences are of type nat ⇒ 'a. However, for stuttering invariant logics, finite sequences can be represented by postfixing them with infinite stuttering of the last state. This follows from the fact that stuttering does not change the validity of a formula. Thus,

finite and inifinite sequences do not need to be separated, and the complexity of the two representations can be avoided.

The Sequence theory contains a framework for mechanising such stuttering invariants logics. The theory abstracts over the underlying state space, represented as a type variable 'a, meaning state-dependent formalisation cannot be mechanised. For example, flexible quantification requires formalisation of equality of two states up to a variable, and can only be defined for a particular Sequence instantiation. Now, a sequence is represented as a function from natural numbers nat to the state space 'a:

types 'a seq = nat ⇒ 'a

This is the same representation as you would often find in papers and textbooks on the subject, enabling a more direct formalisation of the semantics compared to other representations like lazy lists. Standard required operators are defined as follows:

first :: 'a seq ⇒ 'a
first s ≡ s 0

suffix :: 'a seq ⇒ nat ⇒ 'a seq (_ $|_s$ _)
s $|_s$ i ≡ λ n. s (n+i)

tail :: 'a seq ⇒ 'a seq
tail s ≡ s $|_s$ 1

prefix :: 'a seq ⇒ nat ⇒ 'a seq (_[..._])
s[...i] ≡ λ n. if n ≤ i then s n else s i

app :: 'a ⇒ 'a seq ⇒ 'a seq (_ ## _)
w ## s ≡ λ n. if n = 0 then w else s (n-1)

s $|_s$ i returns the suffix of sequence s starting at the i^{th} state, and s[...i] returns the finite prefix of s ending in the i^{th} state. To achieve a type-correct definition, it is made infinite by suffixing the sequence with infinite stuttering of the last state i. A sequence s is finite if it contains an infinite sequence of stuttering steps:

fin :: 'a seq ⇒ bool
fin s ≡ ∃ i. ∀ j≥i. s j = s i

One particular property of a finite sequence is that it has a last element. The index of which is returned by

last :: 'a seq ⇒ nat
last s ≡ LEAST i. ∀ j≥i. s j = s i .

This uses Isabelle's LEAST operator, which returns the smallest i such that ∀ j≥i. s j = s i holds. In other words, the termination step of the finite sequence is returned. It assumes that such an i exists, i.e. fin s.

A special case of a finite sequence is an empty sequence, which only contains stuttering steps, while a non-empty sequence contains a non-stuttering step:

```
empty          :: 'a seq ⇒ bool
empty s        ≡  ∀ i. s i = s 0

notemptyseq    :: 'a seq ⇒ bool
notemptyseq s  ≡  ∃ i. s i ≠ s 0 .
```

For an infinite sequence, the reverse of property fin, must hold:

```
inf    :: 'a seq ⇒ bool
inf s  ≡  ∀ i. ∃ j≥i. s j ≠ s i ,
```

and the following properties hold

lemma `finite_or_infinite:` ∀ s. fin s ∨ inf s
lemma `not_finite_is_infinite:` (¬ fin s) = inf s
lemma `not_infinite_is_finite:` (¬ inf s) = fin s

where unbound variables are free, and can thus be seen as universally quantified. The proofs of these properties explore standard predicate negation over quantifiers.

3.3.1 Stuttering invariance

The key novelty with the Sequence theory, is the treatment of stuttering invariance, which enables verification of stuttering invariance of the operators derived using it. Such proofs require contrasting sequences up to stuttering. Here, Lamport's [120] method is used to mechanise the equality of sequences up to stuttering: he defines the ♮ operator, which collapses a sequence by removing all stuttering steps, except possibly infinite stuttering at the end of the sequence. These are left unchanged. Now, mechanising this operator requires some auxiliary definitions:

```
nonstutseq     :: 'a seq ⇒ bool
nonstutseq s   ≡  ∀ i. s i = s (Suc i) ⟶ (∀ j > i. s i = s j)

stutstep       :: 'a seq ⇒ nat ⇒ bool
stutstep s n   ≡  (s n = s (Suc n))
```

```
nextnat      :: 'a seq ⇒ nat
nextnat s    ≡  if emptyseq s then 0 else LEAST i. s i ≠ s 0

nextsuffix   :: 'a seq ⇒ 'a seq
nextsuffix s ≡  s |_s (nextnat s) .
```

Suc is Isabelle/HOL's successor function for natural numbers. The nonstutseq predicate holds if the sequence only contains stuttering steps at the end, i.e. no stuttering steps except after termination; the stutstep predicate holds if the given state identifier of a sequence is stuttering; nextnat returns the index of the next non-stuttering step. If the sequence is empty, the first element is returned, since the sequence is infinitely stuttering; nextsuffix returns the suffix of sequence s, starting at the first non-stuttering step. The primitive recursive next function finds the n^{th} non-stuttering step:

```
next          :: nat ⇒ 'a seq ⇒ 'a seq
next 0        = id
next (Suc n)  = nextsuffix o (next n).
```

Here, o expresses function composition and id is the identity function. The definition of next follows from Wahab's work in PVS [197]. Finally, the collapse ($\natural$) function is then defined as follows:

```
collapse  :: 'a seq ⇒ 'a seq (♮ _)
♮ s       ≡  λ n. (next n s) 0.
```

Similarity of sequences

Since adding or removing stuttering steps does not change the validity of a stuttering-invarant formula, equality is often too strong, and the weaker equality *up to stuttering* is sufficient. This is often called *similarity* ($\approx$) of sequences in the literature, and is required to show that logical operators are stuttering invariant. This is mechanised as:

```
similar  :: 'a seq ⇒ 'a seq (_ ≈ _)
s ≈ t    ≡  ∀ n. (♮ s) n = (♮ t) n.
```

Due to ease of use, this definition is used, instead of the more general $\natural s = \natural t$, however:

lemma `coleq_seqsim:` $(\natural s = \natural t) = (s \approx t)$

Proof outline. The proof follows directly from the definition of $\natural$ and $\approx$. $\therefore$

The following two theorems are required for the stuttering invariance proofs in the next section. Firstly, if the same element is appended to two similar sequences, then the

resulting sequences are similar:

theorem app_similar: **assumes:** s ≈ t **shows:** (x ## s) ≈ (x ## t)

Proof outline. By case-split of stutstep (x ## s) 0:

(1) when stutstep (x ## s) 0, the assumption implies stutstep (x ## t) 0, and lemma seqsim_stutstep implies (x ## s $|_s$ (Suc 0)) ≈ (x ## s $|_s$ 0) and (x ## t $|_s$ (Suc 0)) ≈ (x ## t $|_s$ 0). The goal follows from the assumption and properties of $|_s$ and ≈ .

(2) when ¬ stutstep (x ## s) 0, ¬ stutstep (x ## t) 0 follows directly. Further, lemma seqsim_notstutstep implies both ((x ## s) $|_s$ (Suc 0)) ≈ nextsuffix ((x ## s) $|_s$ 0) and ((x ## t) $|_s$ (Suc 0)) ≈ nextsuffix ((x ## t) $|_s$ 0). The goal follows from the assumption and standard $|_s$ and ≈ properties, together with nextsuffix distribution over next. ∴

The second theorem is the key theorem for stuttering invariance of transitions. In [144], Merz formalises this theorem as:

theorem sim_step (first attempt): **assumes:** s ≈ t
shows: ∀ n. ∃ m. (s $|_s$n ≈ t $|_s$m) ∧ (s $|_s$(Suc n) ≈ t $|_s$(Suc m)).

and uses it to prove that actions in TLA* are stuttering invariant. However, mechanising it shows that this does not hold. For example, a contradiction is obtained when the (s_n, s_{n+1}) step is stuttering, while the (t_m, t_{m+1}) step is not stuttering. Instead, the following weaker (and more complicated), but correct, theorem is used:

theorem sim_step: **assumes:** s ≈ t
shows: ∀ n. ∃ m. (s $|_s$n ≈ t $|_s$m) ∧
((s $|_s$(Suc n) ≈ t $|_s$(Suc m)) ∨ (s $|_s$(Suc n) ≈ t $|_s$m))

Proof outline. By induction on n:

(1) the base-case is by a case-split on stutstep s 0: stutstep s 0 follows from lemma seqsim_notstutstep, as described in the proof of app_similar above; when ¬ stutstep s 0, notemptyseq s and notemptyseq t follows from stutnempty, seqsim_empty_or_notempty and seqsim_notempty_notempty. Firstly, (s $|_s$ Suc 0) = nextsuffix s follows from notstutstep_nextsuffix1. Secondly, nextsuffix t = t $|_s$ Suc ((nextnat t)-1) follows from nextnat_empty_gzero, seqsim_prev_nextnat and gzero_sucpreveq. The goal then becomes trivial since the assumption and lemma seqsim_suffix_seqsim implies that nextsuffix s ≈ nextsuffix t.

(2) in the step-case the two disjuncts of the second conjunct of the induction hypothesis are proved separately: here, both the $(2a)$: s $|_s$(Suc n) ≈ t $|_s$(Suc m) case and the $(2b)$: s $|_s$(Suc n) ≈ t $|_s$m case are similar to the base case. ∴

3.4 The **Semantics** theory

This section describes a *shallow* embedding of TLA* using Sequence and Intentional. A shallow embedding represents TLA* using Isabelle/HOL predicates, while a *deep* embedding represents TLA* formulas and pre-formulas as mutually inductive datatypes[1]. The choice of a shallow over a deep embedding is motivated by the following factors: a shallow embedding is normally less involved, and existing Isabelle theories and tools can be applied more directly to enhance automation; due to the lifting in the Intentional theory, a shallow embedding can reuse standard logical operators, whilst a deep embedding requires a different set of operators for both formulas and pre-formulas; and finally, since the target is system verification and not meta-properties of TLA*, which requires a deep embedding, a shallow embedding is more fit for purpose.

To mechanise the TLA* semantics, the following type abbreviations are required:

```
types ('a,'b) formfun = 'a seq ⇒ 'b
      'a formula      = ('a,bool) formfun
      ('a,'b) stfun   = 'a ⇒ 'b
      'a stpred       = ('a,bool) stfun.
```

An ('a,'b) formfun is a function over a sequence with state space of type 'a, returning a value of type 'b. A TLA* formula is a sequence predicate, thus of type 'a formula. Similarly, a stfun is a function on a state, while a stpred is a predicate on a state. Now, the always (□) operator has the definition:

```
always :: 'a formula ⇒ 'a formula (□_)
□F     ≡  λ s. ∀ n. (s |_s n) ⊨ F,
```

and, as with the Intentional theory, brackets are used to show the syntax in the lifted world type-class. The ∘ operator generalises TLA's priming operator to formulas. It accepts state functions lifted to sequences, and is thus of the ('a,'b) formfun type:

```
nexts :: ('a,'b) formfun ⇒ ('a,'b) formfun (∘_)
∘F    ≡  λ s. (tail s) ⊨ F
```

This embedding deviates from [143], where state, action and temporal levels are distinct. Thus, before ($) lifts state functions into functions on sequences. Moreover, ∘ is very often used on state functions, thus after is a specialisation of ∘ for state functions, and is equivalent to the priming operator in TLA:

[1] See e.g. [201] for a discussion about deep vs. shallow embeddings in Isabelle/HOL.

```
before  :: ('a,'b) stfun ⇒ ('a,'b) formfun ($_)
$f      ≡  λ s. (first s) ⊨ f

after   :: ('a,'b) stfun ⇒ ('a,'b) formfun (_$)
f$      ≡  ○$f
```

Based on these two definition, the Unchanged state predicate can be lifted to a sequence:

```
unch          :: ('a,'b) stfun ⇒ 'a formula (Unchanged _)
Unchanged f   ≡  f$ = $f
```

The final semantic modality is the action, which uses the Unchanged predicate:

```
action   :: 'a formula ⇒ ('a,'b) stfun ⇒ 'a formula (□[_]_(_))
□[P]_v   ≡  λ s. ∀ i. (s |_s i) ⊨ (P ∨ Unchanged v)
```

The remaining modal operators, like $\Diamond$F and $\Diamond\langle$P$\rangle$_v, are abbreviated from these. Flexible quantification is state-dependent, thus it cannot be defined just yet.

3.4.1 Stuttering invariance

A key feature of TLA*, is that specification at different abstraction levels can be compared. The soundness of this relies on the stuttering invariance of formulas. Since the embedding is shallow, it cannot be showed that a generic TLA* formula is stuttering invariant. However, this section will show that each operator behaves as required with respect to stuttering, which can be used to show stuttering invariance for given specifications.

Stuttering invariance states that if two formulas are similar, then F holds in one iff it holds in the other. Since $\circ$ is generalised to sequence functions, and not just predicates, the definition of stuttering invariance must be generalised to functions:

```
stutinv      :: ('a,'b) formfun ⇒ bool (STUTINV _)
STUTINV F    ≡  ∀ s t. s ≈ t ⟶ (s ⊨ F) = (t ⊨ F).
```

The requirement for stuttering invariance is too strong for pre-formulas. A transition requires two states, thus a stuttering invariant pre-formula cannot define progress. The only place a pre-formula P can be used is inside an action: □[P]_v. To show that □[P]_v is stuttering invariant, is must be shown that a slightly weaker predicate holds for P. For example, if P contains a term of the form ○○Q, then it is not a well-formed pre-formula, thus □[P]_v is not stuttering invariant. This weaker version of stuttering invariance has been named *nearly stuttering invariance*:

```
nstutinv      :: ('a,'b) formfun ⇒ bool (NSTUTINV _)
NSTUTINV F    ≡  ∀ s t. (first s = first t) ∧ (tail s ≈ tail t) ⟶ (s ⊨ F) = (t ⊨ F)
```

and deviates from stutinv by requiring that in the first step either both or none of the sequences changes (to the same!) value, and is strictly weaker than stutinv:

theorem `stut_imp_nstut`: **assumes:** STUTINV F **shows:** NSTUTINV F

Proof outline. The proof reduces to proving that (first s = first t) ∧ (tail s ≈ tail t) implies s ≈ t. This follows from lemma seq_app_first_tail and app_similar. ∴

before is defined for one state (state function), and is therefore stuttering invariant:

theorem `stut_before`: stutinv $F

Proof outline. The proof follows directly from lemma sim_first. ∴

Next, □P must be stuttering invariant. However, it requires that P is stuttering invariant, i.e. stuttering invariance is preserved:

theorem `stut_always`: **assumes:** stutinv F **shows:** stutinv □F

Proof outline. The proof reduces to showing that (s ⊨ □F)=(t ⊨ □F), with the assumptions (A1): s ≈ t and (A2): ∀ s t. s ≈ t ⟶ (s ⊨ F)=(t ⊨ F). Both directions are similar, thus only the ⟶ direction is shown: By the definition of □ the proof reduces to showing (t $|_s$ n) ⊨ F) for an arbitrary but fixed n. (A1) and lemma sim_step implies (s $|_s$ n)=(t $|_s$ n), and the goal then follows from unfolding □ in the assumption ((s $|_s$ n) ⊨ F). ∴

○ F is nearly stuttering invariant if F is stuttering invariant:

theorem `stut_next`: **assumes:** stutinv F **shows:** nstutinv ○F

Proof outline. The proof follows from the definition of stutinv and nstutinv. ∴

Finally, stuttering invariance of the action is shown, which is the most involved proof:

theorem `stut_action`: **assumes:** nstutinv P **shows:** stutinv □[P]_v

Proof outline. The proof reduces to showing that (s ⊨ □[P]_v)=(t ⊨ □[P]_v), with the assumptions (A1): s ≈ t and (A2): ∀ s t. (first s = first t) ∧ (tail s ≈ tail t) ⟶ (s ⊨ P)=(t ⊨ P). Both directions are similar, thus only the ⟶ direction is shown: firstly from (A1), sim_step implies that (A3): (t $|_s$ n) ≈ (s $|_s$ m) and (A4): (((t $|_s$ Suc n) ≈ (s $|_s$ Suc m)) ∨ ((t $|_s$ Suc n) ≈ (s $|_s$ m))). Now, the proof reduces to proving (t $|_s$ n ⊨ P)

$\vee$ (t $|_s$ n $\vDash$ Unchanged v), and is proved with the assumptions (1) s $|_s$ n $\vDash$ P and (2) s $|_s$ n $\vDash$ Unchanged v separately. (1): The proof has two branches where each of the two disjunctions of (A4) are assumed in each of the branch: the first implies t $|_s$ n $\vDash$ P and the second implies t $|_s$ n $\vDash$ Unchanged v. (2): The proof has the same structure as (1), however, both branches imply t $|_s$ n $\vDash$ Unchanged v. $\therefore$

Note, that Merz's paper proof of this theorem [144] is based on the false sim_step(first attempt) theorem above, which results in a shorter and more elegant proof. Finally, the proofs of the abbreviated operators, like F$, follow from the proofs above:

theorem `nstut_after:` nstutinv F$

Proof outline. The proof follows from stut_before and nstut_next. $\therefore$

3.5 The PreFormulas theory

Semantic separation of formulas and pre-formulas requires a deep embedding. However, the PreFormulas theory introduces $\vdash\!\!\sim$

```
PreValid  :: ('w::world) form ⇒ bool (⊢∼ _)
⊢∼ F      ≡  ∀ w. w ⊨ F
```

which is semantically identical to $\vdash$. It's intention is to help users to separate pre-formulas from formulas to achieve correct stuttering-invariant specification.

The theory also proves a large set of theorem about the distribution of before ($_), after (_$) and Unchanged, used to enhance automation. All of these are trivial to prove, and are listed in Appendix A.1. For example, Unchanged distributes over pairs:

theorem `unch_pair:` ⊢∼ Unchanged (x,y) = (Unchanged x ∧ Unchanged y)

which can be used to (automatically) rewrite Unchanged into a "normal form" (together with it's definition): for example, Unchanged (x,y,$\cdots$) into x$ = $x ∧ y$ = $y ∧ $\cdots$.

3.6 The Rules theory

In the Rules theory, the proof system from Figure 2.3 is derived, which is used to mechanise the higher-level rules from Figure 2.2. This theory is still state-independent, thus state-dependent enableness proofs, required for proofs based on fairness assumptions, and flexible quantification, are not discussed here.

3.6.1 Heterogeneous vs. homogeneous proof system

Merz [144] suggest both a *hetereogeneous* and a *homogenous* proof system for TLA*, where the former is shown in Figure 2.3. The homogeneous version eliminates the auxiliary relation $|\sim$, creating a single provability relation $\vdash^h$. This axiomatisation is based on the fact that a pre-formula can only be used via the (sq) rule. To simplify, (sq) is applied to (pax1), ..., (pax5), and (nex),(pre) and (pmp) are changed to accommodate this. For example, (pax2) becomes

$$\vdash^h \Box[\circ(F \Rightarrow G) \Rightarrow (\circ F \Rightarrow \circ G)]_v.$$

Merz argues that while the hetereogenous version is easier to understand, the homogenous system should be better to use for mechanisation and actual verification. In particular, since pre-formulas cannot be separated from formulas in the shallow embedding, a single homogenous proof system whould be preferable. However, the Intentional theory is based on unlifting the world type-class into Isabelle/HOL, and using Isabelle/HOL directly for the proofs. For example, unlifting $\Box$[P]_v creates w $\vDash \Box$[P]_v, which doesn't enable reasoning about P. In fact, P has to be reasoned about separately, thus resulting in a heterogeneous version, which is why heterogeneous is explored.

3.6.2 The basic axioms

Most axioms of Figure 2.3 can be derived directly using the semantic definitions in the Semantic theory. For example,

$$(\mathsf{sq}) \quad \frac{\vdash\!\!\sim P}{\vdash \Box[P]_v}$$

is derived as

theorem sq: **assumes:** $\vdash\!\!\sim$ P **shows:** $\vdash \Box$[P]_v

Proof outline. The proof becomes trivial after unfolding the definitions. ∴

The only axiom that cannot be derived is:

$$(\mathsf{ax3}) \quad \vdash \Box[F \Rightarrow \circ F]_F \Rightarrow (F \Rightarrow \Box F)$$

which provides an induction principle for invariants, and is thus heavily used. The F subscript of (ax3) is a short-hand notation for the free-variables of F, which cannot be computed in a shallow embedding. Now, the free variables of F provide the smallest

possible state function v such that if F holds and v is unchanged then $\circ F$ holds. Thus, (ax3) can be generalised to:

theorem ax3: **assumes:** $\vdash_{\sim} \mathsf{F} \wedge \mathsf{Unchanged\ v} \longrightarrow \circ\mathsf{F}$
shows: $\vdash \Box[\mathsf{F} \longrightarrow \circ\mathsf{F}]\_\mathsf{v} \longrightarrow (\mathsf{F} \longrightarrow \Box\mathsf{F})$

Proof outline. By induction on the sequence. $\therefore$

id returns the complete state function, thus $(*)$: $\vdash_{\sim} \mathsf{F} \wedge \mathsf{Unchanged\ id} \longrightarrow \circ\mathsf{F}$ holds, if stutinv F holds. Thus, in cases where v is not used in the goal of a theorem, $(*)$ can be replaced by stutinv F. This follows from lemma pre_id_unch (see Appendix A.1).

3.6.3 Derived rules

The failure of the deduction theorem [94] for temporal logic has already been discussed. However, Merz defines a variant of this for TLA*:

$$(\vdash \Box F \Rightarrow G) = \big((\vdash F) \Rightarrow (\vdash G)\big)$$

This is proved by derivation over the $\vdash$ and $\vdash_{\sim}$ relations. Further, Deduct is required with the basic axioms for the derivation of many higher-level rules, unless semantic reasoning is used. To derive Deduct, $\vdash$ and $\vdash_{\sim}$ must be given a mutual inductive definitions using the basic axioms (Figure 2.3). Deduct then follows by derivation over these two relations. However, $\vdash$ and $\vdash_{\sim}$ are derived semantically in this mechanisation, thus Deduct cannot be proved, and has to be assumed:

axioms Deduct: $(\vdash \Box\mathsf{F} \longrightarrow \mathsf{G}) = \big((\vdash \mathsf{F}) \longrightarrow (\vdash \mathsf{G})\big)$

Based on these derivations (and axiom), a rule-set, strong enough to derive the higher-level TLA rules for both safety and liveness (see Section 3.7), is derived. Note that rules (STL5) and (STL6) require a semantic derivation, mainly due to the extra assumption of (ax3).

3.6.4 Higher level derived rules

In most verification tasks the low-level rules discussed above are not used directly. Instead the higher level rules (TLA1), (TLA2), (INV1) and (INV2), and specialisation and even higher-level rules for particular domains are applied. Compared to Lamports rules (Figure 2.2), the non-stuttering invariant proofs are separated by the $\vdash_{\sim}$ relation. The following rules are directly applied in the later chapters.

INV1 mechanises Lamports (INV1) rule in TLA*. Here, ∘ replaced the priming operator, and $\leadsto$ separates the pre-formula from formula:

theorem `INV1`: **assumes:** $\leadsto I \wedge [N]\_f \longrightarrow \circ I$ **shows:** $\vdash I \wedge \Box[N]\_f \longrightarrow \Box I$

Proof outline. By induction using ax3, and application of the rules sq and ax4. ∴

invmono specialises INV1 for invariants P, which deviates from the initial state I, of a monolithic specification:

theorem `invmono`: **assumes:** $\vdash I \longrightarrow P$ **and** $\leadsto P \wedge [N]\_f \longrightarrow \circ P$
shows: $\vdash I \wedge \Box[N]\_f \longrightarrow \Box P$

Proof outline. The proof follows directly from INV1. ∴

preimpsplit is used to split the unchanged and computation steps of an action:

theorem `preimpsplit`:
assumes: $\leadsto I \wedge N \longrightarrow Q$ **and** $\leadsto I \wedge$ Unchanged v $\longrightarrow Q$
shows: $\leadsto I \wedge [N]\_v \longrightarrow Q$

Proof outline. The proof follows from the definition of [N]_f. ∴

refstep is used to split the unchanged and computation steps of an action in a refinement (transformation) proof:

theorem `refstep`:
assumes: $\leadsto$ Unchanged v $\longrightarrow$ Unchanged w **and** $\leadsto P \longrightarrow Q \vee$ Unchanged w
shows: $\leadsto [P]\_v \longrightarrow [Q]\_w$

Proof outline. The proof follows from the definition of [N]_f. ∴

refinement1 is used to reduce a temporal level refinement (transformation) goal into a goal comparing the initial states, and a goal comparing the computations:

theorem `refinement1`: **assumes:** $\vdash P \longrightarrow Q$ **and** $\leadsto [A]\_f \longrightarrow [B]\_g$
shows: $\vdash P \wedge \Box[A]\_f \longrightarrow Q \wedge \Box[B]\_g$

Proof outline. The 'initial state' Q of refinement1 is trivial from the first assumption. The action $\Box[B]\_g$ follows from the second assumptions, using lemmas MM8, T3 and T5. ∴

spec_inv2_mono is used strengthen the action N by an invariant J, in the proof of P. Note that there are no restrictions on P, thus this rule can be used in for example invariant and refinement (transformation) proofs:

theorem `spec_inv2_mono`:
assumes: $\vdash I \wedge \Box[N]\_v \longrightarrow \Box J$ **and** $\vdash I \wedge \Box[N \wedge J \wedge \circ J]\_v \longrightarrow P$
shows: $\vdash I \wedge \Box[N]\_v \longrightarrow P$

Proof outline. The proof follows from ax1 and INV2. ∴

inv_join is used to join two invariants of the same specification into one. This helps automate the application of spec_inv2_mono when the specification must be strengthen by several invariants:

theorem `inv_join`: **assumes:** $\vdash P \longrightarrow \Box Q$ **and** $\vdash P \longrightarrow \Box R$
shows: $\vdash P \longrightarrow \Box(Q \wedge R)$

Proof outline. The proof follows from STL5. ∴

inv_case enables case-split directly in the temporal level:

theorem `inv_case`: **assumes:** $\vdash P \longrightarrow \Box(A \longrightarrow B)$ **and** $\vdash P \longrightarrow \Box(\neg A \longrightarrow B)$
shows: $\vdash P \longrightarrow \Box B$

Proof outline. The proof follows from STL5. ∴

3.7 The Liveness theory

As discussed in Section 2.3, liveness properties are verified using weak (WF) or strong (SF) fairness assumptions, together with exploring well-foundedness (LATTICE). The Liveness theory mechanises these proof rules. Proofs using fairness assumptions, require reasoning about *enableness* of actions.

The *Enabled* predicate for TLA action is defined in (2.4) on page 16. In TLA*, actions are replaced by pre-formulas which generalise formulas, thus the enabled predicate has to be extended to pre-formulas, defined over sequences and not pairs of states, and is not discussed in [144]. Now, in TLA, *Enabled* reduces an action predicate into a state predicate, thus in TLA* this must reduce a sequence predicate into a state predicate:

```
enabled1      :: 'a formula ⇒ 'a (Enabled1 _)
Enabled1 F    ≡   λ s. ∃ t. (s ## t) ⊨ F,
```

which states the existence of a sequence t, where F holds when appended to the current state. However, since the state-level is seen as a temporal level in the embedding, enabled lifts enabled1 into sequences, i.e. the temporal level:

enabled :: 'a formula ⇒ 'a formula (Enabled _)
Enabled F ≡ λ s. ∃ t. ((first s) ## t) ⊨ F.

Weak and strong fairness is then defined as follows:

WeakF :: 'a formula ⇒ ('a,'b) stfun ⇒ 'a formula (WF(_)'_(_))
WF(F)_v ≡ ◇□Enabled⟨F⟩_v ⟶ □◇⟨F⟩_v

StrongF :: 'a formula ⇒ ('a,'b) stfun ⇒ 'a formula (SF(_)'_(_))
SF(F)_v ≡ □◇Enabled⟨F⟩_v ⟶ □◇⟨F⟩_v.

Only weak fairness will be discussed, although strong fairness rules has also been derived, and are listed in Appendix A.1. The following lemma is used in the proofs below:

lemma `WF_always`: ⊢ □WF(F)_v = WF(F)_v

Proof outline. The ⟶ direction follows directly from the axiom ax1. The ⟵ direction follows from the rules WF_alt (which follows from E12), MM6 and STL4. ∴

Now, strong fairness deviates from weak fairness by the modalities prefixing Enabled. Thus, if ⟨F⟩_v is always enabled then weak and strong fairness for F is equivalent:

theorem `Enabled_WF_SF`: ⊢ □Enabled⟨F⟩_v ⟶ SF(F)_v = WF(F)_v

Proof outline. The ⟶ direction of the conclusion follows from rule E3, which implies ⊢ □Enabled⟨F⟩_v ⟶ ◇□Enabled⟨F⟩_v. The ⟵ direction applies rule STL4 in addition to rule E3. ∴

The WF1 rule, used to verify leads-to properties from a weak fairness assumption, is mechanised as follows:

theorem `WF1`: **assumes:** ⊩ P ∧ [N]_f ⟶ ○P ∨ ○Q
and: ⊩ P ∧ ⟨N ∧ A⟩_f ⟶ ○Q
and: ⊢ P ⟶ Enabled ⟨A⟩_f
and: ⊩ P ∧ Unchanged f ⟶ ○P
shows: ⊢ □[N]_f ∧ WF(A)_f ⟶ (P ⇝ Q)

Proof outline. By STL4, STL5, T2 and WF_always, the goal is reduced to ⊢ □[N]_f ∧ WF(A)_f ⟶ (P ⟶ ◇Q), and by LT32 it is sufficient to show w ⊨ □P ⟶ ◇Q. Now,

the assumptions STL4 and E3 imply $w \models \Diamond\Box$Enabled $\langle A\rangle_f$, and by the weak fairness assumption and E3, $w \models \Diamond\langle A\rangle_f$ holds. By sq, AA5, AA18, AA24, AA19, E25 and E10, together with the assumption $w \models P \wedge \langle N \wedge A\rangle_f$ holds. The second assumption of WF1 implies $w \models \circ Q$, thus the goal holds by rule E23. $\therefore$

The rule deviates from Lamport's only by the additional $\mid\!\sim P \wedge$ Unchanged $f \longrightarrow \circ P$ assumption as a result of the (indirect) use of ax3, and the syntactic separation of $\mid\!\sim$ from $\vdash$. WF2 is used to derive a fairness assumption from another fairness assumption:

theorem `WF2`: **assumes:** $\mid\!\sim \langle N \wedge B\rangle_f \longrightarrow \langle M\rangle_g$
and: $\mid\!\sim P \wedge \circ P \wedge \langle N \wedge A\rangle_f \longrightarrow B$
and: $\vdash P \wedge$ Enabled $\langle A\rangle_g \longrightarrow$ Enabled $\langle A\rangle_f$
and: $\vdash \Box[N \wedge \neg B]_f \wedge WF(A)_f \wedge F \wedge \Diamond\Box$Enabled $\langle M\rangle_g \longrightarrow WF(M)_g$
shows: $\vdash \Box[N]_f \wedge WF(A)_f \wedge \Box F \longrightarrow WF(M)_g$

Proof outline. By contradiction. $w \models \neg WF(M)_g$ is assumed, and (A1) $w \models \neg\Box\Diamond\langle M\rangle_g$ and consequently (A2) $w \models \Diamond\Box[\neg M]_g$ follows directly from this. The proof follows by showing both (1) $w \models \neg\Box\Diamond\langle B\rangle_f$ and (2) $w \models \Box\Diamond\langle B\rangle_f$. Only the proof of (1) is outlined.

(1) By rule AA26 the proof is reduced to $w \models \Diamond\Box[\neg B]_f$. Now, the first assumption together with (A1), AA25 and STL4 implies $w \models \neg\ \Box\Diamond\langle N \wedge B\rangle_f$. This, together with $w \models \Box[N]_f$, which is the implication assumption of the goal, and the rules MM11, E14, E8 and T2, implies $w \models \Diamond\Box[N \wedge \neg B]_f$. The goal then follows from rule STL6_act, E14 and M6. $\therefore$

The (LATTICE) is derived by directly applying Isabelle/HOL's well-founded relation wf, and has thus been given the name wf_lattice, and follows directly from [143]. This direct application of wf illustrates the advantage of a shallow embedding:

theorem `wf_lattice`: **assumes:** wf r
and: $\forall$ x. $w \models F\ x \rightsquigarrow (G \vee (\exists\ y.\ \#((y,x) \in r) \wedge f\ y))$
shows: $w \models F\ x \rightsquigarrow G$

Proof outline. By induction using Isabelle/HOL's induction rule for wf. The remaining proofs use the rules LT19, LT21, MM4, MM2, MM4b, MM3 and a case-split on $\#((y,x) \in r$. $\therefore$

3.8 The State theory

Since TLA is untyped, and Isabelle/HOL is simply typed, the state space representation is of great importance. Firstly, is strong typing yielded? Here, strong typing

denotes that each variable has a declared type and is only assigned values of that type [63]. Secondly, does the state space representation enable sufficient meta-reasoning to formalise flexible quantification? There are several possible options. A *tuple* representation enables strong typing, but variable access is by index in the tuple, and meta-reasoning is not supported. *Records*, used in e.g. [176, 177], extends tuples by supporting variable access by name. A *hidden*, or abstract state space, supports strong typing and direct variable access, but does not supports meta-reasoning. A *variable-to-value* mapping does not support strong typing, but support meta-reasoning, and is used in the statespace package of Isabelle/HOL.

Ehmety and Paulson [63] uses an abstract type to represented the state space in their Unity mechanisation in Isabelle/HOL, based on [143]. Here, they conclude that a variable-to-value representation may be better, and is first explored in Section 3.8.1, which shows that it is inadequate when the target is system verification. Section 3.8.2 implements the State theory, which follows from [63, 143].

3.8.1 The state space as a variable-to-value mapping

The hidden state space does not support sufficient meta-reasoning to derive the ∃ operator. Merz used this hidden state representation in his Isabelle/HOL embedding [143], hence ∃ is axiomatised and not derived. Here, it is attempted to properly derive the operators whenever possible, which motivates the variable-to-value representation. Strong typing is not supported in such embedding. However, as explained below, it was believed that this could be overcome by projection functions, which further motivated this work. As will become evident below, this is in fact not the case, which caused this representation to be abandoned.

Now, to achieve the embedding, an undefined new type Var is used to represent a variable:

```
typedecl  Var
types     'a state = Var ⇒ 'a
```

Flexible quantification requires equivalence of sequences (equpto), which is defined based on equivalence of two states (state_equpto):

```
state_equpto  :: 'a state ⇒ Var ⇒ 'a state ⇒ bool (_ st=_ _)
s st=v t      ≡  ∃ q. s = t(v := q)

equpto        :: ('a state) seq ⇒ Var ⇒ ('a state) seq ⇒ bool (_ =_ _)
s =v t        ≡  ∀ i. (s i) st=v (t i),
```

where Isabelle's function update t(v := q) equals t for all inputs except v, which is q.

These two definitions are sufficient to mechanise Lamport's ∃ operator:

$$\begin{array}{ll} \mathsf{EEx} & :: \mathsf{Var} \Rightarrow (\text{'a state}) \ \mathsf{formula} \Rightarrow (\text{'a state}) \ \mathsf{formula} \ (\exists\exists \_ : \_) \\ \exists\exists \ x : F & \equiv \ \lambda \ s. \ \exists \ p \ t. \ (s \approx p) \wedge (t =_x p) \wedge (t \vDash F) \end{array}$$

which preserves stuttering invariance:

theorem `stut_EEx`: **assumes:** stutinv F **shows:** stutinv $\exists\exists$ x : F

Proof outline. By standard predicate reasoning following the definition of EEx and stuttering invariance, and symmetry and transitivity of $\approx$. ∴

In a proof, ∃ is reasoned with using either the introduction or elimination rule. As discussed in Section 2.3, these rules are similar to the corresponding rules for rigid quantification. In Isabelle/HOL, ∃ introduction is defined as F x ⟶ ∃ x. F x, and exploits Isabelle's **binder** constructs. This construct implicitly keeps track of bound variables, and ensures capture avoiding substitution.

Now, an Isabelle **binder** has the type ('a ⇒ 'b) ⇒ 'b, hence using **binder** to define EEx, requires the type (Var ⇒ ('a state) formula) ⇒ ('a state) formula. In Isabelle/HOL, the corresponding witness and binding variable are both the same term type, while in ∃ the witness is a state function (('a state) ⇒ 'b), and the binding variable is a variable (Var). Furthermore, representing state functions and variables with a union type, either result in too large a set of hypothesis in the introduction and elimination rules (for practical usage), or the rules are not valid. Thus, the **binder** construct cannot be applied, and substitution of a variable with a state function must be explicitly mechanised to create the ∃ introduction rule.

For this discussion, capture avoidance can be ignored. This could for example been achieved nominally [187, 188]. Substitution of a state function for a variable in a formula (tempsub), is defined by substitution in the sequence of the formula (seqsub), which is defined by substitution for a state (stsub):

$$\begin{array}{ll} \mathsf{stsub} & :: (\text{'a state}) \Rightarrow ((\mathsf{Var} \Rightarrow \text{'a}) \Rightarrow \text{'a}) \Rightarrow \mathsf{Var} \Rightarrow (\text{'a state}) \\ \mathsf{stsub \ s \ f \ x} & \equiv \ \mathsf{s}(x := \mathsf{f \ s}) \end{array}$$

$$\begin{array}{ll} \mathsf{seqsub} & :: ((\text{'a state}) \ \mathsf{seq}) \Rightarrow ((\mathsf{Var} \Rightarrow \text{'a}) \Rightarrow \text{'a}) \Rightarrow \mathsf{Var} \Rightarrow ((\text{'a state}) \ \mathsf{seq}) \\ \mathsf{seqsub \ s \ f \ x} & \equiv \ \lambda \ \mathsf{n. \ stsub \ (s \ n) \ f \ x} \end{array}$$

$$\begin{array}{ll} \mathsf{tempsub} & :: ((\text{'a state}) \ \mathsf{formula}) \Rightarrow \mathsf{Var} \Rightarrow ((\mathsf{Var} \Rightarrow \text{'a}) \Rightarrow \text{'a}) \\ & \qquad \Rightarrow ((\text{'a state}) \ \mathsf{formula}) \ (\_[\_ \mapsto \_]) \\ \mathsf{F}[x \mapsto f] & \equiv \ \lambda \ \mathsf{s. \ (seqsub \ s \ f \ x)} \vDash \mathsf{F} \end{array}$$

The introduction rule for ∃ is then derived using the tempsub:

theorem `eexI:` $\vdash F[x \mapsto f::('a\ state \Rightarrow 'a)] \longrightarrow \exists\exists\ x : F$

Proof outline. By the definition of EEx and substitutions. $\therefore$

At first glance, the problem with the weak typing of the variable-to-value mapping can be overcome by representing the state space ('a) with the union type of all required types, and creating projection functions for each variable: for example instead of using variable x directly, the project $px \equiv \lambda\ s.\ to\_myType(s\ x)$ is applied.

The problem with this is that the proof of eexI, requires that a state function has type (Var ⇒ 'a) ⇒ 'a, and not the more general (Var ⇒ 'a) ⇒ 'b, required to use the projection function px instead of x in a specification. Needless to say, such an approach will be very cumbersome to actually work within a system verification task, and specification alone will be difficult to get correct. This problem is a result of Isabelle/HOL's weak type system, and may be resolved in systems like PVS or Coq, which support dependent types (the type of a projected Var depends on Var), or a type-free logic like Isabelle/ZF. In summary, flexible quantification in TLA cannot be mechanised in a convenient way in a variable-to-value mapping approach, when the motivation is system verification. Since this is the case in this thesis, the variable-to-value mapping was abandoned and the hidden state space approach [63, 143] was instead deployed.

3.8.2 A hidden state space

The idea behind the hidden state space approach [63, 143] is a state space which is defined by it's projections, and everything else is unknown. Thus, a variable is a projection of the state space, and has the same type as a state function. Moreover, strong typing is achieved, since the projection function may have any result type. To achieve this, the state space is represented by an undefined type, which is an instance of the world class to enable use with the Intentional logic:

typedecl state
instance state :: world ..

The proof is achieved by the .. method, since world does not contain any axioms. Since the state space type is defined, the types can be simplified:

types 'a statefun = (state,'a) stfun
statepred = bool statefun
'a tempfun = (state,'a) formfun
temporal = state formula

Reasoning about both enableness of pre-formulas and application of the ∃ elimination rule require the set of free variables. This cannot be found in a shallow embedding, and must be specified by the user for each specification, using the basevars predicate:

```
basevars     :: 'a statefun ⇒ bool
basevars f   ≡   range f = UNIV
```

which states that the state function f comprises the full state space, which means that the range of it is the universal set. For example, basevars (x,y) states that x and y are the base/free variables of the given specification. Note that each variable must be distinct, i.e. basevars (x,x) is false. A consequence of this definition is:

theorem `basevar:` ∀ vs. basevars vs ⟶ ∃ u. vs u = c

Proof outline. This follows from the elimination rule of Isabelle's range, and properties of the universal set UNIV. ∴

The key lemma used in enableness proofs is base_enabled:

theorem `base_enabled:`
assumes: basevars vs **and** ∃ c. ∀ u. vs (first u) = c ⟶ (((first s) ## u) ⊢ F)
shows: s ⊨ Enabled F

Proof outline. By the definition of enableness and the basevars theorem. ∴

Since state meta-reasoning is not possible, the required rules are axiomatised:

axioms

```
eexI:      ⊢ F x ⟶ ∃∃ x. F x
eexE:      ⟦ s ⊨ (∃∃ x. F x); basevars vs;
             ⋀x. ⟦ basevars (x,vs); s ⊨ F x ⟧ ⟹ s ⊨ G ⟧ ⟹ s ⊨ G
eex_stut:  stutinv (F x) ⟹ stutinv (∃∃ x. F x)
history:   ⊢ (I ∧ □[A]_v) = (∃∃ h. $h = ha ∧ I ∧ □[A ∧ h$ = hb]_(h,v)) .
```

eexI is the introduction rule for ∃ and is similar to the Isabelle/HOL introduction rule for ∃. eexE formalises the elimination rule for ∃, and the basevars predicate ensures freeness of variable x. eex_stut states that ∃ preserves stuttering invariance. Finally, history is used to introduce a *history variable* [3]. The validity of eexI and eex_stut are, at least to some point, justified from eexI and stut_EEx theorems in the "variable-to-value mapping" section.

3.9 Reasoning in Isabelle/TLA

A formula F, lifted into the Intensional logic by $\vdash$ F, is verified by unlifting it to the HOL level. Moreover, assumption of $\vdash$ F must be unlifted to enable use at the unlifted HOL level. To achieve this, Intensional contains an introduction (intI) and destruction rule (intD) for lifted formulas, derived as follows:

theorem `intI`: **assumes:** $\bigwedge$ w. w $\vDash$ A **shows:** $\vdash$ A
theorem `intD`: **assumes:** $\vdash$ A **shows:** $\bigwedge$ w. w $\vDash$ A.

The following example shows how these theorems are applied to show modus podens in the lifted logic:

lemma `int_mp`: **assumes:** $\vdash$ A **and** $\vdash$ A $\longrightarrow$ B **shows:** $\vdash$ B.

Proof outline. intI is first applied in a backwards style, which reduces the goal to the unlifted form w $\vDash$ B. intD is then applied to two assumption, thus deriving w $\vDash$ A and w $\vDash$ A $\longrightarrow$ B, and simplication reduces the second assumption to (w $\vDash$ A) $\longrightarrow$ (w $\vDash$ B). From w $\vDash$ A and (w $\vDash$ A) $\longrightarrow$ (w $\vDash$ B), the derived goal w $\vDash$ B trivially holds. $\therefore$

The Isabelle simplifier contains a set of rules of the form A $\equiv$ B, and when applied it rewrites A into B. The following rules turns lifted equality into meta-equiality, so the lifted rule can be treated as a rewrite rule in the simplifier:

theorem `inteq_reflection`: **assumes:** $\vdash$ x=y **shows:** x $\equiv$ y

Since the TLA mechanisation is shallow, an arbitrary (lifted) Isabelle/TLA formula in Isabelle cannot be shown to be stuttering invariant: this must be shown for each specific formula. Furthermore, as explained in Section 3.4, rules like pre_id_unch (see Appendix A.1) requires a formula to be proven stuttering invariant. Now, in Section 3.4 it was shown that each formula "behaved as expected" with respect to stuttering. Thus,

lemmas all_stutinv = stut_action stut_always ...

contains all the "stuttering lemmas and theorems", and can be used together with the simplifier to show that a given formula is stuttering invariant.

To reason in Isabelle/TLA at the unlifted HOL level, TLA operators must be "pushed" as far down a term as possible. Moreover, to enable automation some "normal

form" must be used. For example, Unchanged (x,y), (x,y)$ = $(x,y), (x$,y$) = ($x,$y) and x$ = $x ∧ y$ = $y are all semantically equivalent terms, and there several other equivalent permutations. However, in x$ = $x ∧ y$ = $y, all the $ operators are pushed furthest down the term tree and is thus preferable. Any Unchanged term is rewritten to this form by repetitively applying unch_pair (Section3.5), followed by unfolding the definition of Unchanged (Section3.4).

SectionA.1 of Appendix A list all the lemmas required to simplify $_ and _$ terms. They have the names before_foo and after_foo respectively. For example,

lemma `before_fun2:` ⊢ $(f<x,y>) = f <$x,$y>
lemma `after_const:` ⊢~ (#c)$ = #c

pushes $_ into the arguments of a function, and simplifies a _$ constants, which cannot change by definition. Note, that since _$ is not stuttering invariant, ⊢~ is used instead of ⊢. As above, all the theorems are combined by the lemmas construct:

lemmas all_before = before_const before_fun1 before_fun2 ...
lemmas all_before_eq = all_before[THEN inteq_reflection]

and all_before_eq applies inteq_reflection to all terms, to enable use of the simplifier. Similar lemma lists are developed for the after_foo lemmas.

Since TLA* is a linear temporal logic, ∘ can be pushed down a term in the same way as $_ and _$. In fact, this is a generalisation of pushing _$. However, this assumes that in ∘F, F is not a TLA* operator, with the exception of the form $f, since e.g. ◇∘G and □∘G are not valid TLA* terms. However, for these two cases there are the special cases:

theorem `next_eventually:` **assumes:** stutinv F
shows: ⊢~ ◇F ⟶ ¬F ⟶ ∘◇F
theorem `next_always:` **assumes:** stutinv F
shows: ⊢~ □F ⟶ ∘□F

For the non-TLA* operators (i.e. HOL operators) the similar rules to $_ and _$ are derived and listed in Appendix A.1. The lemmas have the same next_foo, for example:

theorem `next_eq:` ⊢~ ∘(F = G) = ((∘F)=(∘G))

These are combined, and converted into the form used by the simplifier using the next_all and next_eq **lemmas**, similar to all_before and all_before_eq above.

3.10 Summary & discussion

Isabelle/TLA, a mechanisation of the TLA* extension of TLA in Isabelle/HOL, has been described. To achieve this, a generic formalisation of sequences for stuttering invariants logics has been created, and a theory for formalising possible-world semantics has been ported from a previous Isabelle/HOL version. Together, these two theories create a base for mechanising stuttering invariant logics using possible-world semantics.

A shallow embedding of the TLA* operators has been mechanised. Stuttering invariance of these operators is proved, and the TLA* proof system derived. Finally, state representation for these types of logics are discussed, and a problem with a (standard) variable-to-value encoding is illustrated.

The motivation for a shallow, instead of a deep, embedding was based on *ease of use* and *automation*. As discussed in e.g. [201] a shallow embedding is normally less involved than a deep embedding, and since the overall target here is to embed a logic to reason about Hume programs, this is a key feature. For the possible world semantics embedded here, an additional incentive for a shallow embedding is that the existing Isabelle connectives can be re-used for both formulas and pre-formulas, while in a deep embedding different notations are required. TLA syntax, in particular when mechanised, requires a high level of insight and expertise. The introduction of additional syntax for semantically equivalent connectives makes specification even more difficult, and should thus be avoided if possible. Finally, a shallow embedding allows a direct use of existing theories and tools which helps to enhance automation. Since the overall target was Hume program verification, these properties motivated the use of a shallow embedding.

A deep embedding would enable access to the set of free variables, allowing a correct definition of (ax3), and more elegant way of dealing with enableness and ∃ elimination (instead of basevar). Moreover, in a deep embedding, substitution can be defined independently of the the underlying sequence, and capture avoidance would be easier to define, e.g. using Isabelle's nominal datatype [187, 188]. In a deep embedding the whole TLA* logic can be shown to be stuttering invariant, while in a shallow embedding only the operators can be shown. Thus, all specifications have to explicitly prove stuttering invariance (if required), while this becomes implicit in the deep case. Finally, formulas can be semantically separated from pre-formulas in a deep embedding. Now, whether or not these extra features make up for the extra work and extra difficulties with specification (for example, due to the different types of connectives), requires further work. Considering the ultimate goal is verification, a shallow embedding was probably the correct choice.

A mutual inductive definition of $\vdash$ and $\mid\!\sim$, will enable mechanised soundness and

completeness proofs of the TLA* proof system. Moreover, the "temporal deduction theorem" can be derived. However, as above, these are not important criteria for system verification. Here, for example, integration with TLC would be more worthwhile.

In the next chapter, the Hume embedding into TLA is described. It also describes the mechanisation of Hume in TLA$^+$/TLC for model checking and shows a mechanisation of Hume in Isabelle/TLA.

Chapter 4

The Hume coordination layer in TLA

4.1 Introduction

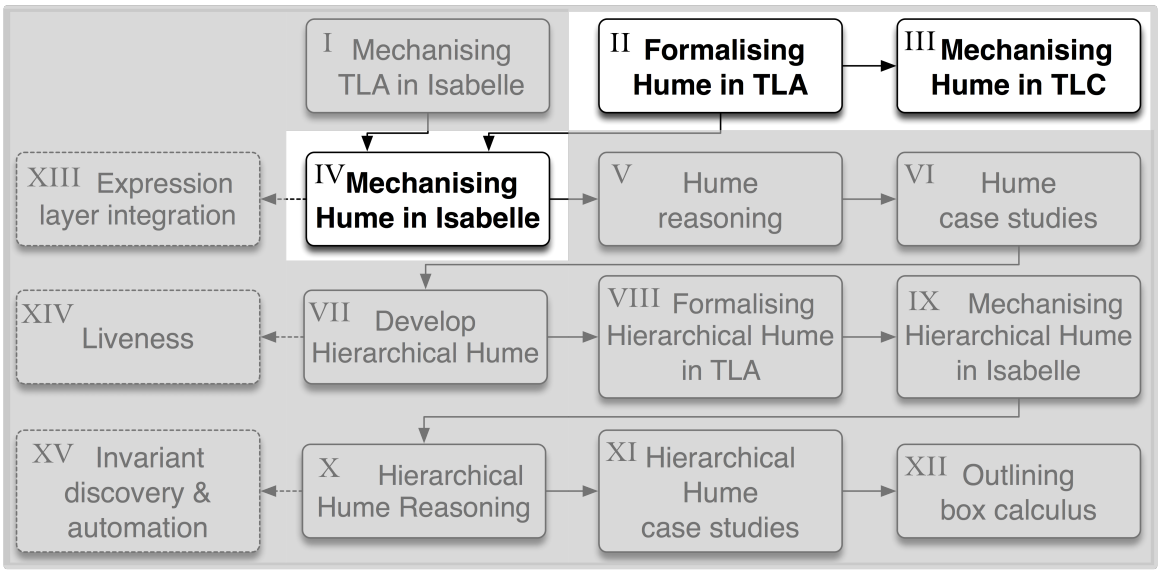


Figure 4.1: Thesis roadmap: Chapter 4

This thesis discusses safety invariants and associated transformations of the Hume coordination layer. The lock step scheduling of Hume boxes introduces two phase: one phase which updates the result buffers; and one phase which updates the wires using the result buffer. Thus, invariants can, and in many cases must, be over both result buffers and wires. The term *wire invariants* will be used for both invariants over the result buffers and wires, unless otherwise is clear from the context.

Tool support, extensibility with liveness and integration with existing expression layer verification tools and techniques are desirable. TLA can capture invariants, transformations and liveness. Tools are supported through the TLC model checker, as well

as theorem proving in Isabelle/TLA.

This chapter discusses the embedding of Hume in TLA, and mechanisation in TLA$^+$ for model checking using TLC and mechanisation in Isabelle/TLA for theorem proving. Figure 4.1 highlights these parts in the roadmap. First however, the motivations behind abandoning an approach extending a translator from HW-Hume into Promela for model checking with Spin is discussed, followed by a discussion of relevant work. Note that Section 9.2 discusses integration with the expression layer in Isabelle/HOL.

4.1.1 Model checking HW-Hume using Spin[1]

In [84] model checking support for Hume was enabled by translating HW-Hume into Promela and model checking with Spin [97]. Since the coordination layer is a finite state automata, this approach was initially considered to be feasible. As part of this PhD, this work was first extended with an assertion language in the Hume source code, which captured both safety and liveness properties. However, this approach was subsequently abandoned for the following reasons:

- the Promela modelling language used by Spin is not expressive enough to capture the Hume expression layer;
- the mechanised expression layer work in Isabelle/HOL [135] cannot be integrated in a direct way;
- the strong dependency between the layers makes it hard to verify the higher-levels, and the strict scheduling makes abstraction difficult;
- the higher-levels tend to be more data-centric and contain more data-centric properties where deductive reasoning is more suitable;
- properties are expressed in a propositional temporal logic, which is often not sufficiently expressive. However, using a Buchi automaton directly in Spin may overcome this particular point;
- transformation verification is not supported;
- experiments suggest that Spin creates more unnecessary extra search states than TLC. However, this may be a result of the embedding of HW-Hume in Promela.

[1] Some of these motivations are discussed in [85].

4.1.2 Relevant work

There are several mechanised logics that have been used to reason at the programming language level. Isabelle/HOL has been used to mechanise Hoare logic [156, 177] and the semantics of for example Java [162] and C [184]. However, due to the concurrency in the Hume coordination layer, none of these are applicable to Hume. Hoare logic is extended with concurrency by the Owicki-Gries method [163], and has been mechanised in [159]. However, in the context of Hume, only wire invariants, and not transformations (or liveness), are supported, while local reasoning is not possible. The latter is solved in *separation logic* [170], which initially extended Floyd-Hoare to pointer programs. It has been used for concurrent systems [161], and mechanised in Isabelle/HOL [198]. In the context of Hume, wire invariants are supported still. There has also been work on data refinement within separation logic [147, 148], however the applicability of this to Hume transformations is a subject of further research.

This chapter describes the embedding of the Hume semantics in TLA. With respect to Hume, the reasoning is thus semantically, since the TLA representation of a Hume program is used, and TLA proof rules are applied directly. In contrast, in a Hoare-based logic the rules are applied syntactically at the programming language level. At the end these are turned into verification conditions (VCs) by a verification condition generator (VCG), and the VCs are verified in a theorem prover. Hence, a Hoare logic must be defined for each programming language, while standard TLA rules are used regardless of which programming language is embedded. As a result, the reuse of tools and integration is better supported in TLA.

Process algebra[2] [18], like CCS [149], CSP [96], the Join calculus [71], the Π-calculus [150] and E-LOTOS [1], is a common technique used to reason about concurrent processes (such as boxes). E-LOTOS is probably the closest to Hume since computation in a process is represented by a functional language. However, communication in a process algebra is stateless, and based purely on processes, while Hume communication is by stateful wires. Thus, a state based representation reduces the semantic gap between Hume and the logic. Moreover, E-LOTOS does not support transformation proofs.

The B-method [8], Event-B [9] and Z [179] extended with the ZRC [40] refinement calculus, could be used for both wire invariants and transformations. They do not have a temporal level, which simplifies both the proofs and representation. However, with the exception of limited leads-to properties in B [10], none of these currently support rich liveness properties. There has also been work integrating such state-based methods with process algebras, like CSP-OZ [69], csp2b [37, 38] and CSP||B [178]. This

[2]Unpublished notes on extending TLA with process algebra constructs are found in [121, 123].

$$
\begin{array}{lrcl}
(1) & \mathcal{S} & \triangleq & (s = \mathit{Execute} \Rightarrow s' = \mathit{Super}) \wedge (s = \mathit{Super} \Rightarrow s' = \mathit{Execute}) \\
(2) & \mathcal{B}_i^e & \triangleq & st_i \neq \mathit{Blocked} \Rightarrow \langle iws_i, res_i, st_i\rangle' = \mathit{execute}(rs_i, iws_i) \\
 & & \wedge & st_i = \mathit{Blocked} \Rightarrow \langle iws_i, res_i, st_i\rangle' = \langle iws_i, res_i, st_i\rangle \\
(3) & Q_i & \triangleq & \forall k \in \mathit{len}(ows_i).\big((res_i)_k = \bot\big) \vee \big((out_i)_k = \bot\big) \\
(4) & \mathcal{B}_i^s & \triangleq & Q_i \Rightarrow \langle ows_i, res_i, st_i\rangle' = \langle nw(res_i, ows_i), \langle \bot, \cdots, \bot\rangle, \mathit{Runnable}\rangle \\
 & & \wedge & \neg Q_i \Rightarrow \langle ows_i, res_i, st_i\rangle' = \langle ows_i, res_i, \mathit{Blocked}\rangle \\
(5) & \mathcal{B}_i & \triangleq & (s = \mathit{Execute} \Rightarrow \mathcal{B}_i^e) \wedge (s = \mathit{Super} \Rightarrow \mathcal{B}_i^s) \\
(6) & I & \triangleq & ws = w_I \wedge s = \mathit{Execute} \\
 & & \wedge & \bigwedge_{i \in BS} \big(st_i = \mathit{Runnable} \wedge res_i = \langle \bot, \cdots, \bot\rangle\big) \\
(7) & H^l & \triangleq & I \wedge \Box[\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i]_{\langle s, ws, res, st\rangle} \\
(8) & H & \triangleq & \exists\ s,st.\ H^l
\end{array}
$$

Figure 4.2: An abstract view of Hume coordination in TLA

results in a state-based computation, and state-less coordination. This is the opposite of the Hume design, where computation is purely functional, and communication is by stateful wires. Thus, a state-based logic like TLA, is more suitable for the Hume communication layer. Since TLA is lifted on top of a predicate logic, it can also be tightly integrated with a mechanisation of the Hume expression layer, as discussed in Section 9.2. With respect to functional languages [129] gives a shallow embedding of a call-by-value functional program, (like ML [151]), while Agerholm's PhD [11] discusses reasoning about functional programs in HOL [80]. Extended-ML [114] is a specification language supporting refinements. As the name implies, it is based on ML [151], and the target of a development is to synthesise ML code. However, Extended-ML is more applicable to the Hume expression layer, than the coordination layer.

4.2 A high-level coordination layer[3]

Figure 4.2 contains a high-level formalisation of the Hume coordination layer. The formalisation has the following assumptions:

- there is a set *BS* of box identifiers, henceforth *boxes*;
- all variables in Figure 4.2, except the scheduling variable s, are tuples of distinct variables;
- the i^{th} tuple-element is projected by a sub-script, e.g. st_i;
- for boxes, tuples are ordered with respect to the box identifier, for example st_i is the state for box $i \in BS$;

[3]The main content of this section has been published in [88].

- there is a tuple ws of wires; each box i has a tuple of input wires iws_i and output wires ows_i;
- feedback loops are allowed, thus a wire in iws_i may also occur in ows_i;
- if for all $i \in BS$, iws_i or ows_i are joined, it equals ws assuming there are no streams and the ordering is ignored;
- a wire in ws may occur in only one iws_i and one ows_i;
- each box i has a state variable st_i and a result buffer res_i;
- for all $i \in BS$ $st_i \in \{Runnable, Blocked, Matchfail\}$;
- res_i has the same number of elements as ows_i.

The definitions in Figure 4.2 then have the following informal meanings:

(1) the scheduler $\mathcal{S}$ alternates between the execute (*Execute*) and super-step (*Super*) phase;

(2) $\mathcal{B}_i^e$ is the box action for an arbitrary box $i \in BS$ in the execute phase. Since unchanged variables must be explicitly specified in TLA, the check that the box is *Runnable* or *Matchfail* is part of the box action. Note that $st_i \in \{Runnable, Blocked, Matchfail\}$ is assumed for all $i \in BS$ (st_i can be set to *Matchfail* by the *execute* function). Hence a box is *Runnable* or *Matchfail* when it is not *Blocked*. If this check succeeds then $execute(rs_i, iws_i)$ will match and consume the required inputs, produce the result output, and set the new state – i.e. *Runnable* if a match succeeds and *Matchfail* if not. This is achieved by the given rule set rs_i, which contains the matches of box $i \in BS$. The *execute* function is for now left undefined. It refers to *execute* in the description of the scheduling algorithm in Section 2.6. If the check fails, then iws_i and res_i are left unchanged;

(3) the Q_i predicate implements the output assertions, and succeeds if, for all corresponding elements k of res_i and ows_i, one of them is empty ($\perp$), that is, either the wire or output buffer is empty. Here, $len(ows_i)$ is the number of elements in tuple ows_i;

(4) $\mathcal{B}_i^s$ represents the super-step phase of a box $i \in BS$. If Q_i holds then the output wire is updated according to $nw(res_i, ows_i)$, which for now, is left undefined. Moreover, the output buffer is set to the empty value $\perp$, and the state is *Runnable*. If Q_i fails, the output buffer's value is left dangling, and the box will be *Blocked* at least until the next super-step, where $\mathcal{B}_i^s$ is re-applied;

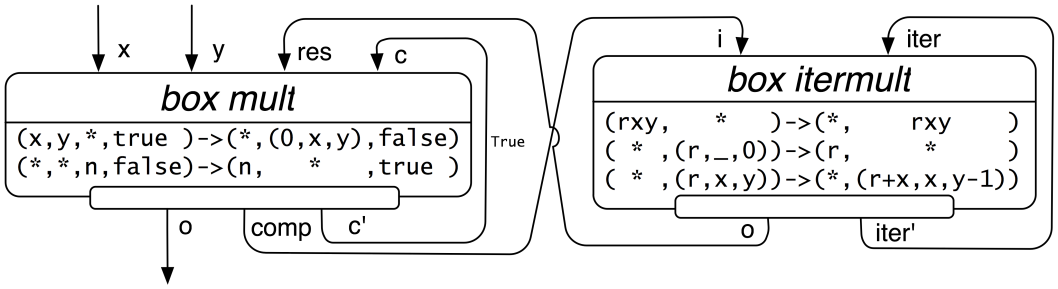


Figure 4.3: Repetition as box iteration

(5) $\mathcal{B}_i$ selects $\mathcal{B}_i^e$ or $\mathcal{B}_i^s$, depending on the current scheduling phase s;

(6) I defines the initial state. In the Hume source code wires can be specified with an initial value, and w_I is a tuple holding all the initial values, depending on the Hume source code: if no initial value is specified for a wire, then the wire equals $\perp$. Moreover, the scheduler is initially in the *Execute* phase, all boxes are initially *Runnable* and the result buffers are initially empty;

(7) H^l is the program specification: the complete next-state action is the conjunction of the scheduler $\mathcal{S}$ and the next actions for all boxes $i \in BS$;

(8) finally, H hides the scheduler s and the box states st, which are required to encode a Hume program in TLA, but not part of the actual program. As will be discussed in Section 4.3, this hiding of variables constrains the possible refinements. Moreover, one could argue that *res* is internal for the specification, and should thus be hidden. However, hiding the result buffer would disable the specification of properties over it, which is desirable. Hence, it is not hidden.

4.2.1 Self-out scheduling

In functional languages repetition is expressed as recursion. Even if the recursion is primitive, meaning here that the program is in the PR-Hume level, resource costing becomes hard. However, Hume can instead represent repetition as box iteration [83, 137] , as shown in Figure 4.3. The `mult` box interfaces the program, and `itermult` computes the result of multiplying the inputs by repeated addition, achieved by "box iteration": the first match copies the input from `itermult.comp` into the feedback loop `iter'` $\rightarrow$ `iter`; the second match is the termination of the iteration, where the result is sent back to `mult.res`; the third match is the step case of the iteration, where the result `r` is incremented by `x`, and `y` is decremented by `1`. Thus, the overall computation increments `x`, `y` times. The `mult` box returns the result from `itermult` on the `o` wire,

and the `c'` $\rightarrow$ `c` wire assures that a new computation is not initialised before the result from the previous is returned.

This example is a FSM-Hume program, which compared to PR-Hume, has more decidable resource properties. However, this iteration introduces an unnecessary scheduling overhead: an iteration of depth N (the initial `y` value), requires $N + 4$ scheduling cycles before the output is on the `o` wire, and in $N + 2$ of these cycles, the `mult` box will be in a *Matchfail* state. This will be the case for all boxes that depend, directly or indirectly, on such an iteration.

To overcome this problem, a more efficient scheduler algorithm, called *self-out* scheduling, has been implemented in both the Hume interpreter and compiler. Empirical studies have suggested an average 14% time saving in the interpreter for programs with box iteration. This algorithm explores the fact that in the usual execution of a box, a *Runnable* box may have either asserted outputs to other boxes and itself, or just to other boxes, or just to itself. If it has asserted outputs just to itself then it can have no impact on the ability of any other boxes to consume inputs. Such boxes have the *self-output* property.

Thus, in principle, such boxes may execute repeatedly until they assert an output for another box, without affecting the overall outcome of program execution, provided there are no strong timing dependencies elsewhere in the program. To achieve this, *Runnable* is divided up into two sub-states:

- *Runnable*: The box has successfully consumed inputs and asserted outputs, and *is not only* writing to internal wires.
- *Selfout*: The box has successfully consumed inputs and asserted outputs, and *is only* writing to internal wires.

Then the staged scheduling strategy is:

```
for ever
    if no box is Runnable
        then for each Selfout box
                execute (box)
        else for each Runnable,Selfout and Matchfail box
                execute (box)
    super step.
```

Figure 4.4 shows the TLA actions for the self-out scheduling. $\mathcal{S}$ and I are unchanged from Figure 4.2. The new definitions have the following meanings:

(1) *SO* holds when no boxes are *Runnable* and is used in the execute phase;

$$
\begin{array}{lrcl}
(1) & SO & \triangleq & \bigwedge_{i \in BS} st_i \neq \mathit{Runnable} \\
(2) & \mathit{emp}(\mathit{nows}) & \triangleq & \forall k \in \mathit{len}(\mathit{nows}).\ \mathit{nows}_k = \bot \\
(3) & P_i & \triangleq & (SO \Rightarrow st_i = \mathit{Selfout}) \wedge (\neg SO \Rightarrow st_i \neq \mathit{Blocked}) \\
(4) & s\mathcal{B}_i^e & \triangleq & P_i \Rightarrow \langle iws_i, res_i, st_i \rangle' = \mathit{execute}(rs_i, iws_i) \\
 & & \wedge & \neg P_i \Rightarrow \langle iws_i, res_i, st_i \rangle' = \langle iws_i, res_i, st_i \rangle \\
(5) & s\mathcal{B}_i^s & \triangleq & \big(Q_i \Rightarrow \langle ows_i, res_i \rangle' = \big\langle nw(iws_i), \langle \bot, \cdots, \bot \rangle \big\rangle \\
 & & & \quad \wedge (\mathit{emp}(\mathit{nows}_i') \Rightarrow st_i' = \mathit{Selfout}) \\
 & & & \quad \wedge (\neg \mathit{emp}(\mathit{nows}_i') \Rightarrow st_i' = \mathit{Runnable})\big) \\
 & & \wedge & (\neg Q_i \Rightarrow \langle ows_i, res_i, st_i \rangle' = \langle ows_i, res_i, \mathit{Blocked} \rangle) \\
(6) & \mathcal{N} & \triangleq & \bigwedge_{i \in BS} \big(s = \mathit{Execute} \Rightarrow s\mathcal{B}_i^e\big) \wedge (s = \mathit{Super} \Rightarrow s\mathcal{B}_i^s) \\
(7) & sH^l & \triangleq & I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle st, s, ws, res \rangle} \\
(8) & sH & \triangleq & \exists\ s, st.\ sH^l
\end{array}
$$

Figure 4.4: Self-output scheduling in TLA

(2) $\mathit{emp}(\mathit{nows})$ holds if all projections of nows equals $\bot$. This is used in the super-step phase to separate *Selfout* state from *Runnable*;

(3) P_i is separated out from (4) for purely aesthetic reasons;

(4) $s\mathcal{B}_i^e$ defines the execute phase for an arbitrary box $i \in BS$. It behaves as specified above, by executing only *Selfout* boxes when SO holds, and as in lock-step scheduling when it does not hold;

(5) $s\mathcal{B}_i^s$ defines the box action in a super-step. Here, the tuple nows_i, which is contained in ows_i, holds all the wires that are *not* wired back to box $i \in BS$. $\mathit{emp}(\mathit{nows}_i')$ holds if all output values are on the wires wired back to i. If the box only writes to internal wires, i.e. $\mathit{emp}(\mathit{nows}_i')$, then it is in a *Selfout* state;

(6) $\mathcal{N}$ is separated out from (7) for purely aesthetic reasons;

(7) sH^l updates H^l with the new definitions;

(8) sH updates H with sH^l.

The TLA specification of self-output scheduling is more complex than lock-step scheduling, thus reasoning with self-output scheduling is more involved and complex than lock-step scheduling. However, Theorem 4.1 states that self-output scheduling is a refinement of lock-step scheduling. Thus, all properties of a program under lock-step scheduling, consequently hold for the same program under self-output scheduling. As a result, it is sufficient to reason using lock-step scheduling, and self-output scheduling is ignored henceforth.

$\langle a, b, c\rangle$	:	the list consisting of a, b and c
$\langle\rangle$	:	the empty list
$L_1 \cdot L_2$	:	appends list L_2 at the end of list L_1
$hd(L)$	:	the head of list L, e.g. $hd(\langle a, b, c\rangle) = a$
$tl(L)$	:	the tail of list L, e.g. $tl(\langle a, b, c\rangle) = \langle b, c\rangle$
$len(L)$	:	the length of list L, e.g. $len(\langle a, b, c\rangle) = 3$
if e_1 **then** e_2 **else** e_3	:	standard conditional expression
case $e_1 \rightarrow e'_1 \ \square \ \cdots \ \square$ **else** $\rightarrow \ e'_n$	:	standard case expression

Table 4.1: Underlying operators

Theorem 4.1. *Self-output scheduling is a refinement of lock-step scheduling:* $sH \Rightarrow H$

Proof outline. The refinement mapping replaces s by the state function $\overline{s} \triangleq s$ and st_i with $\overline{st_i} \triangleq$ **if** $st_i = Selfout$ **then** $Runnable$ **else** st_i. The proof is by case analysis on s, followed by case analysis on st_i in the $s = Execute$ phase, and Q_i in the $s = Super$ phase, for an arbitrary but fixed box $i \in BS$. $\therefore$

A detailed proof of Theorem 4.1 is shown in Appendix B.

4.3 Refining the coordination layer

A mechanisation of Hume programs in TLA for actual system verification, requires more details than shown in Figure 4.2. Moreover, the atomicity of the execute phase is too abstract when discussing transformation verification. Firstly, details are given of the operators, which are given a more operational definition, by replacing quantifiers by recursive functions. This is followed by a discussion on how streams are handled, before the atomicity of the expression layer is removed.

4.3.1 Refining operations

To define the operations a list notation is used, as shown in Table 4.1. Note that since TLA is type-free, this notation also serves for tuples, and (potentially infinite) sequences, although the append operator $\cdot$ assumes that L_1 is finite. In addition to list operators, **if** and **case** expressions are required.

execute of $\mathcal{B}^e_i$ in Figure 4.2 is first refined. This requires a formalisation of pattern matching in TLA. Firstly, two disjointed sets are assumed: Id holding box, wire, constructor and function identifiers, ranged over by id or i for boxes ($BS \subseteq \mathsf{Id}$); and Var

holding variables, ranged over by x. A single pattern p is of the form

$$p \in \{\_*, *, x, id, \_\} \tag{4.1}$$

As specified in Figure 2.8 (on page 35), c is an enumeration type. Moreover, id has to have arity 0, i.e. it is a constant. The rule-set rs_i for a box $i \in BS$ is a list of pattern lists and expression pairs, written $p \rightarrow e$, with the following projections and definitions:

$$\begin{aligned} rs_i &\triangleq \langle (p_1 \rightarrow e_1), \cdots, (p_n \rightarrow e_n) \rangle \\ patt(p \rightarrow e) &\triangleq p \\ exp(p \rightarrow e) &\triangleq e \end{aligned}$$

Pattern matching for a single pattern-value pair is then defined as

$$pm_1(p, iw) \triangleq p \in \{\_*, *\} \vee (p \in \{x, \_\} \wedge iw \neq \bot) \vee (p = id \wedge id = iw)$$

A list of patterns matches if all pattern elements match. Pattern matching over a list of patterns and a list of values, assumed to be of equal length, is then defined by a primitive recursive function:

$$pm(patt, iws) \triangleq patt = \langle\rangle \vee \Big(pm_1\big(hd(patt), hd(iws)\big) \wedge pm\big(tl(patt), tl(iws)\big) \Big)$$

Following a match, a consumption of the inputs are inferred. Here, all wires except those with a $*$ in the corresponding pattern are set to the empty value $\bot$:

$$\begin{array}{lll} consume(rs, iws) \triangleq & & \\ \quad \textbf{case} & iws = \langle\rangle & \rightarrow \langle\rangle \\ \quad \square & iws \neq \langle\rangle \wedge hd(rs) \neq * & \rightarrow \langle\bot\rangle \cdot consume\big(tl(rs), tl(iws)\big) \\ \quad \square & \textbf{else} & \rightarrow \langle hd(iws)\rangle \cdot consume\big(tl(rs), tl(iws)\big) \end{array}$$

The final part of *execute*, after the pattern matching and input consumption, is executing the expression layer. However, since this is purely an expression layer feature, this discussion is delayed to Section 4.4 and Section 4.5. For now, the undefined function *run* represents this. Finally, *execute* is thus defined as:

$$
\begin{array}{lll}
execute(rs, iws) \triangleq & & \\
\quad \textbf{case} \quad rs = \langle\rangle & & \rightarrow \langle iws, \langle \bot, \cdots \bot \rangle, Matchfail \rangle \\
\quad \square \quad rs \neq \langle\rangle \wedge pm\big(patt\big(hd(rs)\big), hd(iws)\big) & & \rightarrow \left\langle \begin{array}{l} consume\big(patt\big(hd(rs)\big), iws\big), \\ run\big(exp\big(hd(rs)\big), iws\big), \\ Runnable \end{array} \right\rangle \\
\quad \square \quad \textbf{else} & & \rightarrow execute(tl(rs), iws)
\end{array}
$$

It attempts to pattern match each match, and *Matchfail*s if all matches fail.

The super-step action $\mathcal{B}_i^s$ consists of two operations: Q_i which asserts the outputs; and nw which returns the new wire value. Q_i is defined by a bounded universal quantifier over the length of the output wire list. This is given a more operational recursive definition in the ao function:

$$ao(res, ows) \triangleq res = \langle\rangle \vee \big(hd(res) = \bot \vee hd(ows) = \bot\big) \wedge ao\big(tl(res), tl(ows)\big)$$

The nw function updates each output wire recursively, with either the corresponding result buffer, or wire, depending on which, if any, are empty $\bot$:

$$
\begin{array}{lllll}
nw(res, ows) \triangleq & \textbf{case} & ows = \langle\rangle & & \rightarrow \langle\rangle \\
& \square & ows \neq \langle\rangle \wedge hd(res) \neq \bot & & \rightarrow \langle hd(res)\rangle \cdot nw\big(tl(res), tl(ows)\big) \\
& \square & \textbf{else} & & \rightarrow \langle hd(ows)\rangle \cdot nw\big(tl(res), tl(ows)\big)
\end{array}
$$

Finally, the new box action for box i in the super-step phase $\mathcal{B}_i^s$ is refined to:

$$
\begin{array}{lll}
\mathcal{B}_i^s \triangleq & \big(ao(res_i, ows_i) & \Rightarrow \langle ows_i, res_i, st_i\rangle' = \langle nw(res_i, ows_i), \langle \bot, \cdots \bot \rangle, Runnable\rangle\big) \\
\wedge & \big(\neg ao(res_i, ows_i) & \Rightarrow \langle ows_i, res_i, st_i\rangle' = \langle ows_i, res_i, Blocked\rangle\big)
\end{array}
$$

4.3.2 Open systems: representing streams

Hume's finite state machine based design makes it well suited for implementing control systems. Most control systems are reactive by nature, thus communication with the outside world is a paradigm. A Hume program communicates values from and to the outside world by (input and output) streams. The I/O has not been given formal semantics yet. However, both the interpreter and compiler handle this as a special box in the coordination layer, and the same approach is followed here.

The environment, representing the outside world, can either be part of the system or an assumption of the system. The latter approach will form an *assume-guarantee* approach where the environment E is an assumption of the program H. TLA contains a special operator for such specifications, which helps avoiding circular reasoning when

conjoined [6, 5]:

$$E \xrightarrow{+} H.$$

This is read as "*if E holds for the first n steps, then H holds the first $n+1$ steps*". Now, since a stream is represented as a box, the first approach, where it is part of the system, is more natural. Let $\mathcal{E}_0$ represent the action that handles *all* the streams. H_e extends H_l from Figure 4.2 by conjoining the next action with $\mathcal{E}_0$:

$$H_e \triangleq I \wedge \Box[\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i \wedge \mathcal{E}_0]_{\langle s,ws,res,st \rangle}.$$

Now, $\mathcal{E}_0$ has the informal meaning:

$$\begin{array}{rll} \mathcal{E}_0 \triangleq & (s = \mathit{Execute} & \Rightarrow 1.\text{"}\mathit{Consume\ output\ stream\ wires}\text{"}) \\ \wedge & (s = \mathit{Super} & \Rightarrow 2.\text{"}\mathit{Update/write\ input\ stream\ wires}\text{"}) \end{array}$$

Both input consumption (1) and output update (2) can be deterministic or non-deterministic: the environment may always consume/update wires, or it may chose whether or not to consume/update wires. Let w denote the input or output wire, and T the type of w. The deterministic and non-determinstic cases are then formalised as:

	update	*consumption*
deterministic	$\exists v \in T : w' = v$	$w' = \bot$
non-deterministic	$\exists v \in T \cup \{\bot\} : w' = v$	$w' = w \vee w' = \bot.$

Let *in_ws* be the set of wires connected to input streams, and *out_ws* be the set of wires connected to output streams, such that $\mathit{in\_ws} \subset \mathit{ws}$ and $\mathit{out\_ws} \subset \mathit{ws}$[4]. $\mathcal{E}_0$ is then formalised as follows:

$$\begin{array}{rll} \mathcal{E}_0 \triangleq & (s = \mathit{Execute} & \Rightarrow \forall w \in \mathit{out\_ws} : w' = w \vee w' = \bot) \\ \wedge & (s = \mathit{Super} & \Rightarrow \forall w \in \mathit{in\_ws} : \exists v \in T \cup \{\bot\} : w' = v) \end{array}$$

4.3.3 Sequential box execution

The final refinement splits the atomic execute phase into a sequential execution of the boxes. This is the same level of abstraction as used in the Hume structural operational semantics [112], for the round-robin box execution. This concretisation of H from Figure 4.2 has two major impacts:

[4]The $\subset$ and $\subseteq$ set operators are abused here, since *ws*,*out_ws* and *in_ws* are tuples. Note that a wire cannot be wired directly between an input and output stream, thus $\subset$ and not $\subseteq$ is used.

1. program independence of box atomicity, which simplifies transformation proofs;
2. wire consumption atomicity is broken, which complicates wire invariants over several wires consumed by different boxes.

Formalising sequential box execution requires the introduction of a program counter *pc*, and an update of the scheduling action $\mathcal{S}$, the environment $\mathcal{E}_0$ and the full next action $\Box[\cdots]_{\langle s,ws,res,st\rangle}$. Let $|BS|$ be the cardinality of the set BS of all boxes in program H. Now, in the $s = \mathit{Execute}$ phase, $\mathcal{S}$ should change s to *Super* when all boxes have executed, thus the execute phase is extended from 1 to $|BS|$ steps. Moreover, a box $i \in BS$ should only attempt to execute *once* per execute phase, achieved by *pc* which holds the identifier of the box to be executed. Let $\prec$ be a well-founded partial order on $BS \cup \{env\}$, where *env* is a constant identifying the "environment box". The function *first_box* returns the smallest box $i \in BS \cup \{env\}$ according to $\prec$, while *last_box* returns the largest box, which exists since $BS \cup \{env\}$ is finite. The function *next_box(pc)* returns the box immediately following *pc* according to $\prec$, unless *pc* is *last_box*:

$$
\begin{array}{lcl}
\mathit{first\_box} & \triangleq & \epsilon\, pc \in BS \cup \{env\}.\ \forall p \in BS \cup \{env\}.\ pc \neq p \Rightarrow pc \prec p \\
\mathit{last\_box} & \triangleq & \epsilon\, pc \in BS \cup \{env\}.\ \forall p \in BS \cup \{env\}.\ pc \neq p \Rightarrow p \prec pc \\
\mathit{next\_box(pc)} & \triangleq & \\
\quad \textbf{if}\ pc = \mathit{last\_box} & & \\
\quad\ \textbf{then}\ \mathit{last\_box} & & \\
\quad\ \textbf{else}\ \epsilon\, p \in BS \cup \{env\} - \{pc\}.\ \forall q \in BS \cup \{env\} - \{pc, p\}.\ pc \prec p\ \wedge\ p \prec q &&
\end{array}
$$

where ϵ is Hilbert's choice operator[5], and the $\cup$ and $-$ set operators have the same precedence and are read left to right. These functions ensure that each box is executed once and only once. $\mathcal{S}_{seq}$, the scheduler for sequential box execution, is updated to ensure that all boxes are executed before the super-step:

$$
\begin{array}{rcll}
\mathcal{S}_{seq} & \triangleq & (s = \mathit{Execute} \wedge pc' \neq \mathit{last\_box} & \Rightarrow s' = \mathit{Execute}) \\
& \wedge & (s = \mathit{Execute} \wedge pc' = \mathit{last\_box} & \Rightarrow s' = \mathit{Super}) \\
& \wedge & (s = \mathit{Super} & \Rightarrow s' = \mathit{Execute})
\end{array}
$$

$\mathcal{N}_{seq}$ replaces $\bigwedge_{i \in BS} \mathcal{B}_i$ (in H_l) of Figure 4.2:

$$
\begin{array}{rcll}
\mathcal{N}_{seq} & \triangleq & \big(s = \mathit{Execute} & \Rightarrow (pc \in BS \Rightarrow \mathcal{B}^e_{pc}) \\
& & & \wedge \big(\bigwedge_{i \in BS - \{pc\}} \langle iws_i, res_i, st_i\rangle' = \langle iws_i, res_i, st_i\rangle \big) \\
& & & \wedge\ pc' = \mathit{next\_box(pc)}\big) \\
& \wedge & (s = \mathit{Super} & \Rightarrow (\textstyle\bigwedge_{i \in BS} \mathcal{B}^s_i) \wedge pc' = \mathit{first\_box}\big)
\end{array}
$$

[5] $\epsilon\ x.\ P(x)$ denotes an x such that $P(x)$ holds.

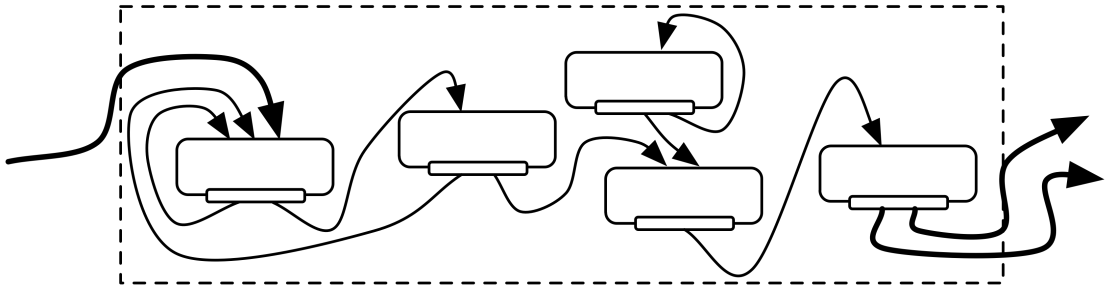


Figure 4.5: Program with "internal wires" hidden.

The $pc \in BS$ guard before the box action $\mathcal{B}^e_{pc}$ is necessary since the environment is executed as a box, while $\mathcal{E}$ extends $\mathcal{E}_0$ to reflect sequential execution:

$$\begin{array}{rll} \mathcal{E} \triangleq & (s = \mathit{Execute} \wedge pc = \mathit{env} & \Rightarrow \forall w \in \mathit{oout\_ws} : w' = w \vee w' = \bot) \\ \wedge & (s = \mathit{Execute} \wedge pc \neq \mathit{env} & \Rightarrow \forall w \in \mathit{ows} : w' = w) \\ \wedge & (s = \mathit{Super} & \Rightarrow \forall w \in \mathit{in\_ws} : \exists v \in T \cup \{\bot\} : w' = v) \end{array}$$

The complete program with sequential execution H_{seq} is then defined as:

$$\begin{array}{rcl} H^l_{seq} & \triangleq & I \wedge (pc = \mathit{first\_box}) \wedge \Box[\mathcal{N}_{seq} \wedge \mathcal{S}_{seq} \wedge \mathcal{E}]_{\langle s,ws,res,st,pc\rangle} \\ H_{seq} & \triangleq & \exists\!\!\exists\ \mathit{st,s,pc}.\ H^l_{seq} \end{array} \tag{4.2}$$

Is lock-step scheduling preserved?

With the current set of free (not hidden) variables (*res* and *ws*), lock step scheduling is not preserved. This follows from the split of atomicity of wire consumption in the execute phase (1 on page 74 of this section) and the fact that wires are not $\exists\!\!\exists$-bound. Informally, since wires are 1-1 relationship, boxes are without side-effects, and output wires are not written to before the super-step, this should hold. However, these properties cannot be captured by the model without $\exists\!\!\exists$-binding the wires and result buffers, which would not correctly reflect a Hume program. A compromise could be to $\exists\!\!\exists$-bind all wires not connected to a stream and look at them as internal, as shown by the thin wires of Figure 4.5. Additional constraints are: (A1) all inputs are wired to the same box; (A2) the result buffer *res* is hidden; (A3) the "input box" *last* is scheduled last ($\forall i \in BS.\ i \preceq \mathit{last}$); (A4) the environment $\mathcal{E}$ is replaced by $\mathcal{E}^l$ which is executed in parallel with *last*. Let *int_ws* be the internal wires. H and H_{seq} are then updated as follows:

$$H^{int} \triangleq \exists\ st,s,res,int\_ws.\ H^{l}$$
$$H^{int}_{seq} \triangleq \exists\ st,s,pc,res,int\_ws.\ I \wedge (pc = first\_box) \wedge \Box[\mathcal{N}_{seq} \wedge \mathcal{S}_{seq} \wedge \mathcal{E}^{l}]_{\langle s,ws,res,st,pc\rangle}$$

Theorem 4.2. *H^{int}_{seq} is a refinement of H^{int}: $H^{int}_{seq} \Rightarrow H^{int}$*

Proof outline. The proof requires the introduction of a history variables st^h, res^h and int_ws^h for *st*, *res* and *int_ws*. The refinement mapping formed by the state functions $\overline{st}$, $\overline{res}$ and $\overline{int\_ws}$ are equal to *st*, *res* and *int_ws* respectively, except for all execute phases but the last ($s = Execute \wedge pc \neq last$), where they are equal to st^h, res^h and int_ws^h.

The proof is similar to the proof of Theorem 4.1, with an additional case split on *pc*. The super-step phase is unchanged from Theorem 4.1, thus is trivial. In the execute-phase *s* is only changed when $pc = last$. Moreover, the refinement mapping $\overline{st}$, $\overline{res}$ and $\overline{int\_ws}$ and the definition of the st^h, res^h and int_ws^h ensures that all internal variables (that are found in H^{int}) are unchanged for all execute-step phases except when $pc = last$. Finally, only $\mathcal{E}^l$ and *last* can by definition update the free wires. However, this is, by (*A3*) and (*A4*), in the last step when $pc = last$. Thus, all steps in the execute phase, except when $pc = last$, are stuttering steps in the abstract model (H^{int}). By definition each box is executed once and only once in the sequential scheduling, and each box is unchanged. Thus, by the definition of $\overline{st}$, $\overline{res}$ and $\overline{int\_ws}$, the execute step where $pc = last$ simulates the single execute step of H^{int}. ∴

Nonetheless, this embedding disables reasoning about properties that are not on wires connected to streams, which is not feasible. So it is not used. Now, the only property verified so far is that the self-output scheduling preserves lock-step scheduling. Since sequential scheduling does not refine lock-step scheduling, this can no longer be assumed. However, self-out scheduling can be updated to sequential self-out scheduling in the same way as lock-step scheduling, and Theorem 4.1 replaced by:

Theorem 4.3. *Sequential self-output scheduling is a refinement of sequential lock-step scheduling.*

Proof outline. The proof is similar to the proof of Theorem 4.1, with an additional case split on *pc* in the execute phase. ∴

4.4 Hume in TLA$^+$ (TLC) for model checking

Since system verification is the motivation behind the work here, Hume is given a shallow mechanisation in TLA$^+$, thus disabling meta properties of the embedding. In

$$S(cond) \triangleq \text{IF } s = Execute \ \wedge \neg cond \text{ THEN } s' = Super \text{ ELSE } s' = Execute$$

$$pm1(p, iw) \triangleq (p = Ignore) \vee (p = Var \wedge iw \neq Bot) \vee (p = iw)$$
$$pm[patt \in Seq(Pattern), iws \in Seq(Lift(Dom))] \triangleq$$
$$\quad Len(patt) \neq 0 \wedge Len(iws) \neq 0 \Rightarrow$$
$$\quad\quad pm1\big(Head(patt), Head(iws)\big) \wedge pm\big[Tail(patt), Tail(iws)\big]$$

$$ao1(res, ow) \triangleq (res = Bot) \vee (ow = Bot)$$
$$ao[res, ows \in Seq(Lift(Dom))] \triangleq$$
$$\quad Len(res) \neq 0 \wedge Len(ows) \neq 0 \Rightarrow$$
$$\quad\quad ao1\big(Head(res), Head(ows)\big) \wedge ao\big[Tail(res), Tail(ows)\big]$$

$$nw1(res, ow) \triangleq \text{IF } res = Bot \text{ THEN } ow \text{ ELSE } res$$
$$nw[res, ows \in Seq(Lift(Dom))] \triangleq$$
$$\quad \text{IF } Len(res) \neq 0 \wedge Len(ows) \neq 0$$
$$\quad\quad \text{THEN } \langle\rangle$$
$$\quad\quad \text{ELSE } nw1\big(Head(res), Head(ows)\big) \cdot nw\big[Tail(res), Tail(ows)\big]$$

Figure 4.6: TLA$^+$ generic actions

all examples, natural numbers and sequences of natural numbers, are sufficient types. Thus, the *Naturals*, *Sequence*, and *TLC* need to be imported to the model. Note that $\exists$ is not supported by TLC, so s and st are not hidden. Now, all mechanised Hume programs require the following constants:

CONSTANTS *Bot, Dom, Ignore, Var*
CONSTANTS *Runnable, Matchfail, Blocked, Execute, Super.*

Bot represents the empty value $\perp$; *Dom* is the union-type of all types used in a program, which is *Nat* in all Hume examples here. *Bot* can be defined as

$$Bot \triangleq \text{CHOOSE } x : x \notin Dom$$

where CHOOSE is Hilbert's ϵ operator in TLA$^+$. However, *Bot* has to be given a value in the TLC configuration anyway, so this is unnecessary. A type T is lifted to the bottom type by *Lift*

$$Lift(T) \triangleq T \cup \{Bot\}$$

which is required by result buffers and wires; *Ignore* captures $*$ and $\_*$ patterns; while *Var* captures a variable x and $\_$ in patterns. Since all constants are contained in *Dom*,

the pattern type *Pattern* is

$$Pattern \triangleq \{Ignore,\ Var\} \cup Dom;$$

the remaining constants are as in the discussion above. Moreover, each program contains a scheduler s and program counter pc:

VARIABLES $s,\ pc$

Figure 4.6 mechanises the the scheduler S ($\mathcal{S}_{seq}$), pattern matching: *pm1* (pm_1) and *pm*, output assertion *ao* and wire update function *nw*, together with auxiliary functions *ao1* and *nw1*. *Head* relates to the *hd* operator and *Tail* to the *tl* operator of Table 4.1. The *cond* argument of S holds if $pc' \neq last\_box$. Since this is a shallow embedding, the condition has to be given as an argument.

4.4.1 An example embedding

```
stream input from "std_in";  stream output to "std_out";
box inc
  in(x::word 8) out(x'::word 8)
match
  x -> x+1;
wire inc (input)(output);
```

Figure 4.7: A simple Hume example

Figure 4.8 shows the mechanisation of the Hume program listed in Figure 4.7: the program consists of one box `inc`, with an input wire *w1* from standard input; and an output wire $w2$ connected to standard output; and `inc` increments the input by 1. In addition to the wires, the box state *inc_st* and result buffer *inc_res* for `inc` are the state components specific to this program:

VARIABLES $w1,\ w2,\ inc\_st,\ inc\_res$
CONSTANTS $penv,\ pinc$.

PC is a constant set replacing BS, with respect to updating pc. Here, it is defined as $PC \triangleq \{penv, pinc\}$. Moreover, only natural numbers (Hume `word` type) are used, thus $Dom \triangleq Nat$.

In Figure 4.8, *inc_exe* defines $\mathcal{B}^e_{pinc}$ in TLA$^+$. However, for simplicity, the *pc* check is moved into the box actions. Pattern matching is achieved using TLA$^+$'s

$$
\begin{array}{lcl}
\mathit{inc\_exe} & \triangleq & \mathit{pc} = \mathit{pinv} \Rightarrow \mathit{pc}' = \mathit{penv} \\
 & & \wedge\ \mathit{inc\_st} \neq \mathit{Blocked} \Rightarrow \text{CASE } \mathit{pm1}(\mathit{Var}, w1) \ \rightarrow\ w1' = \mathit{Bot} \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{inc\_st}' = \mathit{Runnable} \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{inc\_res}' = w1 + 1 \\
 & & \qquad\qquad\qquad\qquad\quad \square\ \text{OTHER} \qquad \rightarrow\ w1' = w1 \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{inc\_st}' = \mathit{Matchfail} \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{inc\_res}' = \mathit{Bot} \\
 & \wedge & (\mathit{pc} \neq \mathit{pinc} \vee \mathit{inc\_st} = \mathit{Blocked}) \Rightarrow \text{UNCHANGED } \langle\ \mathit{inc\_st},\ \mathit{inc\_res},\ w1\ \rangle \\
\mathit{inc\_sup} & \triangleq & \text{IF } \mathit{ao1}(\mathit{inc\_res}, w2) \\
 & & \quad \text{THEN } w2' = \mathit{nw1}(\mathit{inc\_res}, w2) \\
 & & \qquad \wedge\ \langle\ \mathit{inc\_res},\ \mathit{inc\_st}\ \rangle' = \langle\ \mathit{Bot},\ \mathit{Runnable}\ \rangle \\
 & & \quad \text{ELSE UNCHANGED } \langle\ \mathit{inc\_res},\ w2\ \rangle\ \wedge \mathit{inc\_st}' = \mathit{Blocked} \\
\mathit{inc} & \triangleq & (s = \mathit{Execute} \Rightarrow \mathit{inc\_exe})\ \wedge\ (s = \mathit{Super} \Rightarrow \mathit{inc\_sup}) \\
E & \triangleq & s = \mathit{Execute} \Rightarrow \mathit{pc} = \mathit{penv} \Rightarrow \mathit{pc}' = \mathit{pinc} \wedge (w2' = w2 \vee w2' = \mathit{Bot}) \\
 & & \qquad\qquad\qquad \wedge\ \mathit{pc} \neq \mathit{penv} \Rightarrow w2' = w2 \\
 & \wedge & s = \mathit{Super} \quad \Rightarrow (\exists v \in \mathit{Lift}(0..254) : w1' = v) \wedge \mathit{pc}' = \mathit{pinc} \\
I & \triangleq & s = \mathit{Execute} \wedge \mathit{pc} = \mathit{pinc} \wedge \langle\ \mathit{w1},\ \mathit{w2}\ \rangle = \langle\ \mathit{Bot},\ \mathit{Bot}\ \rangle \\
 & \wedge & \langle\ \mathit{inc\_st},\ \mathit{inc\_res}\ \rangle = \langle\ \mathit{Runnable},\ \mathit{Bot}\ \rangle \\
v & \triangleq & \langle s, \mathit{pc}, w1, w2, \mathit{inc\_res}, \mathit{inc\_st} \rangle \\
H & \triangleq & I \wedge \square[\mathit{inc} \wedge E \wedge S(\mathit{pc}' \neq \mathit{penv})]_v \\
sN & \triangleq & (0..255) \\
\mathit{Inv1} & \triangleq & \mathit{w2} \neq \mathit{Bot} \Rightarrow \mathit{w2} \in \mathit{Nat}
\end{array}
$$

```
SPECIFICATION H
CONSTANTS Bot = bot
          Ignore = ignore
          ...
          Nat <- sN
INVARIANTS Inv1
```

Figure 4.8: TLA$^+$ definition and TLC configuration file (boxed) for example

CASE expression, which is equivalent to nested IF-THEN-ELSE expressions. Note that UNCHANGED $v \triangleq v' = v$, and there is only one input (and output), hence *pm1* (*ao1*/*nw1*) are used instead of the sequence *pm* (*ao*/*nw*) functions; *inc_exe* and *inc* defines $\mathcal{B}^s_{pinc}$ and $\mathcal{B}_{pinc}$ in TLA$^+$ respectively; E defines $\mathcal{E}$ in TLA$^+$. For simplicity, the $\mathit{pc}' = \mathit{first\_box}\ (\mathit{pinc})$ is part of E in the super phase; I is the initial state; v the set of variables in the program; and H is the full program specification.

The box at the bottom right of Figure 4.8 outlines the specification file required to model check the program with TLC. Firstly, it specifies the specification (H) and assigns values to all the constants. All infinite sets has to be restricted to a finite subsets, thus the defined *sN* is used instead of *Nat*. Note, since a box increments the input by 1, E can only update $w1$ to a maximum of 1 less then the largest value of *sN*, i.e. 254 (the largest 8 bit number is 255). This is actually a bug in `inc`, and the box should be changed to reflect this. For example, by replacing `x+1` by the "passive

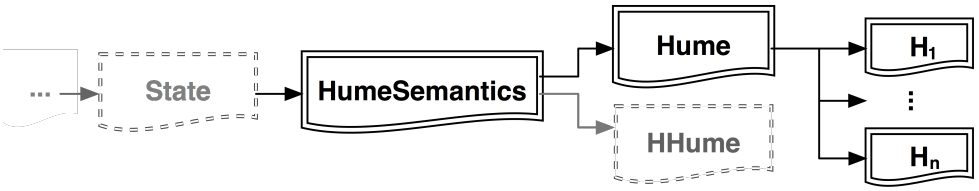


Figure 4.9: Isabelle/TLA/Hume theory structure*

expression" `if x < 255 then x+1 else x`. However, the example is kept as simple as possible since it serves merely as an illustration. The program contains one invariant, *Inv1* which states that $w2$ is a natural number (unless it is empty).

4.5 Hume in Isabelle/TLA for theorem proving

As in the TLA$^+$ mechanisation and for the same reason, Hume is given a shallow mechanisation in Isabelle/TLA. Here, a shallow embedding entails less work in the actual mechanisation and more direct use of existing Isabelle tools and theorems. The types are also given a shallow embedding by using Isabelle/HOL types directly, since the Hume type system is not the focus of the thesis. This is also the case in the expression layer mechanisation, briefly discussed in Section 9.2. To avoid duplicating the expression layer mechanisation, Isabelle/HOL represents this layer directly: primary Hume operators, like `+` and `if-then-else` are represented using corresponding Isabelle/HOL operators (+ and if-then-else); user-defined Hume functions are represented using the Isabelle/HOL function package. This pragmatic approach increases the semantic gap, thus increasing the possibility of errors. However, both the Hume expression layer and Isabelle/HOL are in the ML family [151], hence this gap should not be too large. Finally, note that the expression layer mechanisation has been achieved in parallel and was not ready when this work started, thus it has not been used. However, Section 9.2 shows how to integrate the two layers in Isabelle/HOL. For simplicity, s and st are not ∃ bound here. Sections 7.2 and 7.3, discusses ∃ binding within the context of transformations.

Figure 4.9 shows the theory structure for the Hume mechanisation: it is based on the State theory discussed in the previous chapter; HumeSemantics contains definitions and theorems used by both Hume and Hierarchical Hume (HHume described in Chapter 6); the Hume theory contains definitions and lemmas specific for Hume, which are used for all Hume programs mechanised, illustrated by $\mathsf{H}_1, \cdots, \mathsf{H}_1$. The HumeSemantics and Hume theories are discussed separately and followed by an example of a mechanisation. The mechanised theorems and lemmas are listed in Appendix A.

4.5.1 The HumeSemantics theory

Tuple projections

Tuples of differently typed elements are heavily used in Hume. Since the State theory imposes strong typing, tuples are also required to mechanise both box result buffers and wires. Isabelle has built in polymorphic functions fst(x,y)=x and snd(x,y)=y for pairs. Thus, the second element of a triple t is accessed by fst(snd t). Projections are used to mechanise both programs and properties. To simplify, projection functions for tuples up to eleven elements are created (eleven is the largest sized tuple required). Since tuples are not an algebraic type, each projection has to be defined for each size of a tuple, where for example thd4 is the projection of the third element of a quadruple, defined as:

```
thd4     :: ('a * 'b * 'c * 'd) ⇒ 'c
thd4 q   ≡ case q of (x,y,z,zz) ⇒ z ,
```

Hume values

Both a wire and an element of the result buffer may be empty, which is represented by the existing Isabelle/HOL type

datatype 'a option = None | Some 'a,

where None is used to represent an empty value. intoVal turns a value into an option value; toVal projects the value (of a non-empty) option value; while the isVal predicate holds if the given value is not empty:

```
intoVal          :: 'a ⇒ 'a option
intoVal v        ≡   Some v
toVal            :: 'a option ⇒ 'a
toVal (Some v)   ≡   v
isVal            :: 'a option ⇒ bool
isVal None       ≡   False
isVal (Some v)   ≡   True
```

Since a variable in TLA is a function on the underlying state, a Hume value HVal is an option type state function:

types 'a HVal = ('a option) statefun

For example, a wire w of integer type has the type int HVal. Lifting functions for $_ and _$ into Hume values help simplifying specifications:

```
toBeforeVal  :: 'a HVal ⇒ 'a tempfun (@_)
@v           ≡  toVal<$v>
toAfterVal   :: 'a HVal ⇒ 'a tempfun (_@)
v@           ≡  toVal<v$>
```

Pattern matching

A Hume pattern is defined using an algebraic type

datatype 'a pattern = PIgnore | PConsumeIgnore | PConsume | PConst 'a,

where, with respect to (4.1), PIgnore relates to * ; PIgnoreConsume to _* ; PConsume combines x and _ ; and PConst *id* relates to *id*. PMatch implements pm_1, which accepts a pattern and an input (wire) value. The function is defined by pattern matching on the input 'a pattern type:

```
PMatch                    :: 'a pattern ⇒ 'a option ⇒ bool
PMatch PIgnore iw         = True
PMatch PConsumeIgnore iw  = True
PMatch PConsume iw        = (isVal iw)
PMatch (PConst p) iw      = (isVal iw ∧ toVal iw = p)
```

pm is defined by the conjunction of all PMatches for each match, since due to the typing, a pattern is represented as a tuple, which is not algebraic. Note that the option and not the HVal type is used. This is since the HVal type is lifted to the Intensional logic, and functions in this logic are defined unlifted, and then lifted by enclosing the arguments using <· · ·>.

Wire assertion and updates

As with pattern matches, only ao_1 and nw_1 can be mechanised due to the strong typing, and *ao* and *nw* for a given match, are conjunctions of all the ao_1's and nw_1's. assertOut mechanises ao_1 by pattern matching of the first input:

```
assertOut              :: 'a option ⇒ 'a option ⇒ bool
assertOut None ow      = True
assertOut (Some v) ow  = (ow = None),
```

while new_wire mechanises nw_1 in a similar way:

```
new_wire            :: 'a option ⇒ 'a option ⇒ 'a option
new_wire None ow    = ow
new_wire v ow       = v.
```

4.5.2 The Hume theory

The Hume theory mechanises the scheduler and box states. The scheduling state is defined by an enumeration type

datatype sch_state = Execute | Super.

Each program has a scheduler s, which is a state function of the sch_state type

s :: sch_state statefun .

As in TLA$^+$, the scheduler $\mathcal{S}_{seq}$ cannot be directly implemented due to the shallow embedding. Moreover, due to lifting into Intensional, it cannot update the scheduling variable s either. Instead, S is a function, accepting a scheduling value and a condition ($pc \neq last\_box$) and returns the new scheduling variable:

```
S                :: sch_state ⇒ bool ⇒ sch_state
S Execute True   = Execute
S Execute False  = Super
S Super    _     = Execute
```

A box's state is defined by an enumeration type

datatype box_state = Runnable | Matchfail | Blocked

and a box state variable thus has the box_state statefun type:

types boxstate = box_state statefun

4.5.3 An example embedding

The example shown in Figure 4.7 is used as illustration here as well. Firstly, PC is an enumeration type, and contains the `inc` box (pinc) and the environment box (penv), while pc is a PC state function

```
datatype PC = pinc | penv
pc :: PC statefun .
```

S' specialises the scheduler S for this particular program:

```
S'  :: temporal
S'  ≡   s$ = S<$s, $pc ≠ #penv>
```

The two wires, w1 and w2, of the program are both 16 bit Hume `word`s, which are represented using natural numbers nat in Isabelle:

```
w1   :: nat HVal
w2   :: nat HVal .
```

The state inc_st and result buffer inc_res of the `inc` box have the types:

```
inc_st    :: boxstate
inc_res   :: nat HVal.
```

The box actions for the execute and super-step phase are represented by inc_exe and inc_sup, respectively. To ease the reading, the main body of inc_exe is separated into inc_body. inc is the complete action for the `inc` box:

```
inc_exe     :: temporal
inc_sup     :: temporal
inc_body    :: temporal
inc         :: temporal.
```

inc_exe first checks if it is its turn to execute, i.e. if $pc = #pinc. If this holds, the program counter is set to the next box pc$ = #penv. This is followed by a check of whether the box is executable (not *Blocked*) or not. If it is executable, then inc_body holds. In both checks, in a failure the "owned" variables must be explicitly set to be unchanged. Note that the two checks cannot be combined since pc must be updated whether or not the box is executable when it is the box's turn to execute:

```
inc_exe ≡
      ($pc = #pinc ⟶ pc$ = #penv
                   ∧ ($inc_st ≠ #Blocked ⟶ inc_body)
                   ∧ ($inc_st = #Blocked ⟶ Unchanged (w1,inc_st,inc_res)))
   ∧ ($pc ≠ #pinc ⟶ Unchanged (w1,inc_st,inc_res)).
```

The body of a box consists of a set of nested if-then-else expressions, where each represents a match, followed by the last statement, which handles the case where all patterns fail to match with the input. A match updates the result buffer, state and consumes the input. In the `inc` box there is only one match, creating the following box body:

```
inc_body ≡    if PMatch< #PConsume , $w1>
              then inc_res$ = intoVal<Suc<@w1>>
                 ∧ (inc_st, w1)$ = (#Runnable, #None)
              else  (w1, inc_st, inc_res)$ = ($w1, #Matchfail, #None).
```

The super-step phase is mechanised using one if-then-else expression, which is the case for all programs and boxes:

```
inc_sup ≡    if assertOut<$inc_res , $w2>
             then (inc_res, inc_st)$ = (#None, #Runnable)
                ∧ w2$ = nwire<$inc_res, $w2>
             else  (inc_res, w2, inc_st)$ = ($inc_res, $w2, #Blocked).
```

Finally, inc finds the correct action, depending on the scheduler s:

```
inc  ≡    ($s = #Execute ⟶ inc_exe) ∧  ($s = #Super ⟶ inc_sup).
```

The environment env resets pc to the first box (pinc) in addition to simulating the streams. Since it behaves as a box, it only executes when it is its turn:

```
env  ::   temporal
env  ≡    ($s = #Execute ⟶ (if  $pc = #penv
                                then (w2$ = $w2 ∨ w2$ = #None) ∧ pc$ = $pc
                                else  w2$ = $w2))
     ∧    ($s = #Super ⟶ (∃ v. w1$ = #v) ∧ pc$ = #pinc).
```

Note that the deterministic update is achieved in Isabelle/HOL by exploring typing information, that is enclosing #v by `intoVal`: ∃ v. w1$ = intoVal<#v >. The initial step is defined as:

```
init ::   temporal
init ≡    $s = #Execute ∧ $pc = #pinc
     ∧    $(w1,w2) = #(None,None)
     ∧    $(inc_st, inc_res) = #(Runnable, None),
```

and the full specification of the `inc` example becomes :

```
program ::   temporal
program ≡    init ∧ □[S' ∧ inc ∧ env]_(s,pc,w1,w2,inc_st,inc_res)
```

4.6 Summary & discussion

This chapter has formalised the Hume coordination layer in TLA, and mechanised Hume in TLA$^+$ for model checking, and Isabelle/TLA for theorem proving. The latter is henceforth called Isabelle/Hume. The main focus on the remaining chapters is on theorem proving in Isabelle/Hume. This is mainly to support integration with the higher Hume levels, which requires higher order logic, and integration with the expression layer in Isabelle/HOL, discussed in Section 9.2. Chapter 9 also discusses a liveness extension. Finally, model checking will eventually suffer from the *state-space explosion* problem, and ways of avoiding this are not discussed further, due to the theorem proving focus. Note that TLC has been very useful in achieving the Hume

embedding in TLA.

The next chapter discusses verification of Hume wire invariants and program transformation in TLA, and implements tactics to automate reasoning in Isabelle/Hume. The approach is illustrated by model checking and theorem proving case-studies.

Chapter 5

Verification of Hume programs

"Beware of bugs [in the code]; I have only proved it correct; not tried it."

– Donald Knuth

5.1 Introduction

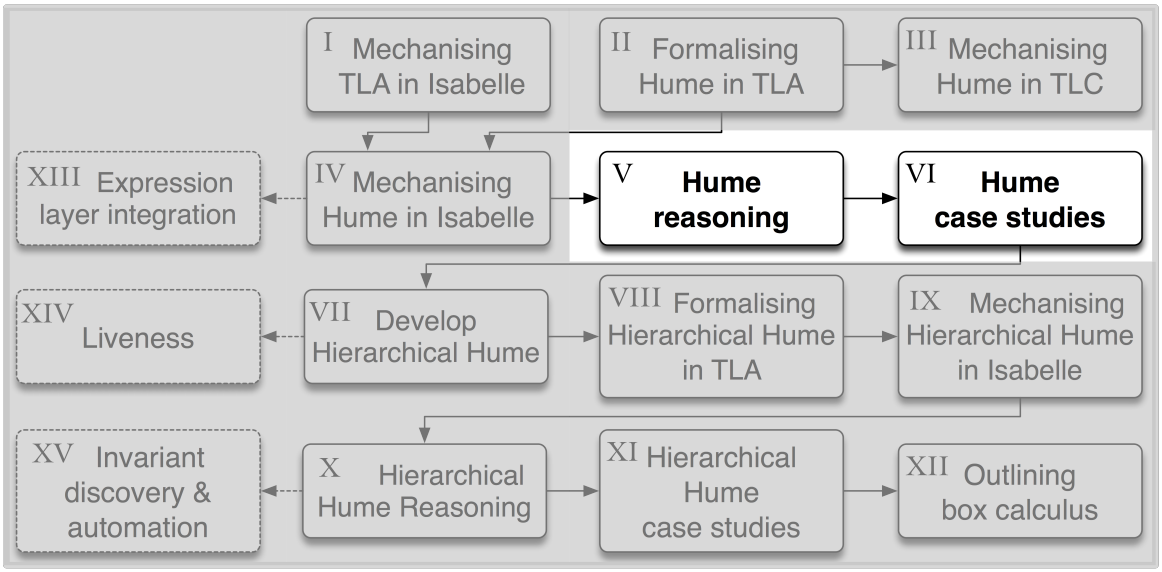


Figure 5.1: Thesis roadmap: Chapter 5

This chapter discusses the verification of Hume invariants and transformation, and Figure 5.1 highlights the parts of the roadmap which are implemented here. Generic mechanised reasoning in Isabelle/Hume is first discussed, followed by a discussion of invariant verification, and the mechanisation of this reasoning in Isabelle/Hume. This is followed by a discussion of transformation verification. The approaches are then illustrated by case-studies in Section 5.5. The main contributions of this chapter are the

development and mechanisation of the reasoning approach for invariants, the transformation verification using TLA, and the case-studies. The work presented here is novel with respect to the application of TLA at the programming language level. All mechanised Isabelle/Hume theorems and lemmas are listed in Appendix A.2 and A.3.

5.2 Isabelle/Hume reasoning

Tuple projections and Hume types

Tuples are heavily used within the Isabelle/Hume embedding, in particular to represent the result buffer of a box with more than one output. There are two different ways to rewrite a tuple: firstly, a lemma like

lemma `tup_proj4:`
$$(x = (a,b,c,d)) = (fst4\ x = a \wedge snd4\ x = b \wedge thd4\ x = c \wedge for4\ x = d)$$

is proved for all sized tuples; secondly, the following lemmas illustrate how one particular value is projected from a tuple, and is proved for all possible projections of all sized tuples:

lemma `tup_lup8:` $thd4\ (a,b,c,d) = c$
lemma `tup_lup9:` $for4\ (a,b,c,d) = d$

Proof outline. All the tup_projN and tup_lupN lemmas are proved automatically by the built-in Isabelle/HOL tools using the tuple definition. ∴

The @ operator was introduced to simplify specification of properties and programs. However, when reasoning it is easier to work with @ unfolded. Thus, @v and v@ are automatically rewritten into Some $v and Some v$.

Pattern matching

The pattern matching function PMatch is defined using Isabelle/HOL's function for primitive recursion. Isabelle automatically generates a set of rewrite rules from this, which are given to the simplifier. Here, the _ and *id* patterns (PMatch PConsume iw) are rewritten to isVal iw, and a constant (PMatch (PConst p) iw) is rewritten to isVal iw ∧ toVal iw = p. This is exploited when reasoning in the execute phase.

Output wires

The output wires are updated in the super-step phase. In a proof obligation, 'the given' is the program and 'the goal' is the property. As discussed below, the proof always requires a case split on assertOut. Thus, the key problem is how to verify the update of output wires, i.e. nwire. There are three main lemmas:

lemma `nwireE`: $[\![x = \mathsf{nwire}\ A\ B;\ x = A \Longrightarrow P;\ x = B \Longrightarrow P]\!] \Longrightarrow P$
lemma `nw1`: $\mathsf{isVal}\ x \Longrightarrow \mathsf{nwire}\ x\ y = x$
lemma `nw2`: $\neg\mathsf{isVal}\ x \Longrightarrow \mathsf{nwire}\ x\ y = y$

Proof outline. All lemmas are proved by induction on x. ∴

nwireE is a (standard) elimination rule for nwire. It turns an assumption w$ = nwire f<$res> $w into two subgoals: one with with assumption w$ = f<$res> and one with the assumption w$ = $w. This is the most general rule. By applying this rule to all nwire, all possible combinations are enumerated. However, there are cases where a property is defined over several wires, and the property depends on whether all or none of the wires are written. For such cases, the "blind" application of nwireE is not sufficient. Moreover, it may enumerate too many unnecessary subgoals. For example, if the property to be verified is over one wire, and the result buffer is an eleven-element tuple, then 2^{11} (2048) sub-goals are created while two are sufficient. In this instance a case-analysis on isVal<f<$res>> followed by application of nw1 and nw2 creates only the two required sub-goals. Moreover, let P be an invariant over two wires, $P(\$w, \$ww)$, where g<$res> is the projection for the second wire. Furthermore, assume that $P(f<\$res>, g<\$res>)$ is already verified. In the step case of an inductive invariant proof, $P(\$w, \$ww)$ is assumed, and $P(w\$, ww\$)$ must be shown. Now, a "blind application" of nwireE induces, for example, the sub-goal $P(f<\$res>, \$ww)$, which cannot be proved from the givens. This kind of invariant requires the verification of isVal<f<$res>> = isVal<g<$res>>, followed by an application of nw1 and nw2. This will create two sub-goals, $P(\$w, \$ww)$ and $P(f<\$res>, g<\$res>)$, where the givens can be applied directly. This is discussed further in Section 5.3.

The scheduler

The Hume theory contains the scheduling specific theories, i.e. the scheduler s. As discussed below, all the proofs are by case-analysis of s, exploring the enumeration type. Thus, no particular proofs are required.

5.2.1 Isabelle/Hume tactics developed

Common reasoning patterns and rule applications are encapsulated in tactics. These tactics exploit the shallow embedding, and attempt to be as light-weight as possible. The rewriting of goals and assumptions is performed by the built-in Isabelle *simplifier*, controlled by limiting the set of rewrite rules used. The tactics here are used in higher-level tactics described later, where control of the sub-goals is required. Thus, all the tactics accept a parameter goal specifying the sub-goal to which the tactic should be applied:

unlift_tac goal is not Hume specific, i.e. it can be used for any Isabelle/TLA term. It unlifts an Intensional-lifted formula to a HOL formula, i.e. $\vdash$ A and $\nvdash$ A becomes w $\vDash$ F, where w is bound by the HOL meta-level universal quantifier $\bigwedge$;

unchanged_tac goal is not Hume specific either. It turns Unchanged (x_1,$\cdots$, x_n) into $x_1\$ = \$x_1 \wedge \cdots\ x_n\$ = \$x_n$;

simplify_hume_tac goal pushes $\circ$, \$,@ and $\vDash$ as far down the term tree as possible, and attempts to combine them. For example, w $\vDash$ $\circ$(F $\vee$ \$G) becomes (w $\vDash$ $\circ$F) $\vee$ (w $\vDash$ G\$);

tuple_project_tac goal simplifies the tuples as illustrated by Lemma tup_proj4 in a largest tuple first order (11 first, then 10 and so on). The ordering is required since for example thd3 (a,b,c,d,e) can be rewritten to (c,d,e);

tuple_lookup_tac goal applies the tuple look-up rewrites as illustrated by Lemmas tup_lup8 and tup_lup9 in the same order as above.

5.3 Invariant verification

5.3.1 Overview

An invariant is either over one or more result buffer projections (of the same box), or one or more wires. Let P, K and L be wire predicates. Their validity may depend on other predicates of other wires. Figure 5.2 classifies invariants based on their dependency. Here, it is assumed that the wire predicate P is on one wire w, where box A is the source of the wire, K is a predicate on wire b of the same box, and L a predicate on wire c from box C. There is then a separation of four different cases:

(D1) $P(w)$ can be proved directly from A, without any strengthening;

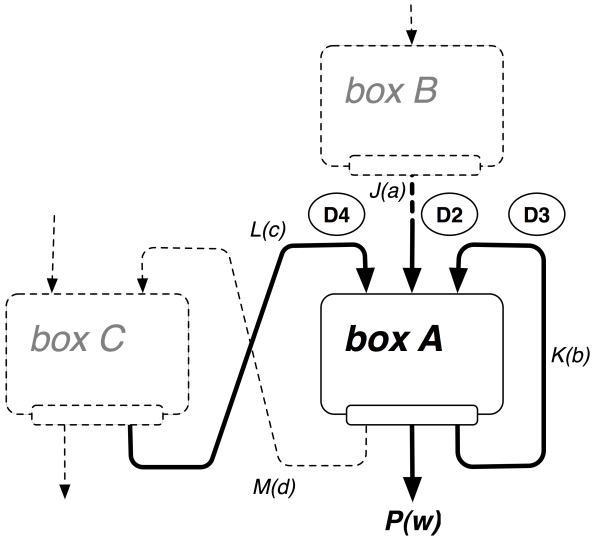


Figure 5.2: The types of invariant and dependencies

(D2) $P(w)$ must be strengthened by $J(a)$ from box B. The result is used with A to prove $P(w)$. Note, there is no direct or indirect dependency from A required to prove $J(a)$;

(D3) $K(b)$ is verified, where b is a feedback wire on box A;

(D4) $L(c)$ from box C is verified. However, $M(d)$ from A is required to prove $L(c)$. Note, that this nesting may involve several $(C_1, C_2, \cdots)$ boxes, and (3) is a special case of (4) where wire c equals wire d and predicate L equals predicate M.

Further, let $f(res_A)$ be the projection of the result buffer that corresponds to wire w, $g(res_A)$ be the projection of the result buffer corresponding to wire b, $h(res_A)$ corresponding to wire d, $i(res_B)$ wire a of the result buffer of box B and $j(res_C)$ wire c of the result buffer of box C.

A result buffer projection and a wire may be empty. Moreover, in the execute phase the result from the computation is stored in the result buffer res_A, and in the superstep phase $f(res_A)$ attempts to copy to w. Thus, the proof of (D1) type invariants first requires the proof of $\Box\Big(f(res_A) \neq \bot \Rightarrow P\big(f(res_A)\big)\Big)$, which is then used to prove the main property $\Box\big(w \neq \bot \Rightarrow I(w)\big)$.

In (D2) style invariants, $\Box\Big(f(res_A) \neq \bot \Rightarrow P\big(f(res_A)\big)\Big)$ must be strengthened by $\Box\big(a \neq \bot \Rightarrow J(a)\big)$. This invariant is proved independently of box `A`.

For (D3) style invariants, $\Box\big(b \neq \bot \Rightarrow K(b)\big)$ must be verified. However, this cannot be proved in the sequential way as in (D1) and (D2) since the loop introduces a mutual dependency: $K\big(g(res_A)\big)$ depends on $K(b)$ (execute phase) and $K(b)$ depends

on $K\big(g(res_A)\big)$ (super-step phase). Thus, the result buffer and wire have to be proven at the same time, meaning the two invariants are combined into:

$$\Box\Big(\big(g(res_A) \neq \bot \Rightarrow K\big(g(res_A)\big)\big) \ \wedge \ \big(b \neq \bot \Rightarrow K(b)\big)\Big). \tag{5.1}$$

(D4) is a generalisation of (D3), where the feedback is via one or more other boxes. In this case all the involved wires and result buffers must be captured by the invariant. For the example in Figure 5.2, this becomes:

$$\Box\left(\begin{array}{lllll} & \big(j(res_C) \neq \bot \Rightarrow L\big(j(res_C)\big)\big) & \wedge & \big(c \neq \bot \Rightarrow L(c)\big) \\ \wedge & \big(h(res_A) \neq \bot \Rightarrow M\big(h(res_A)\big)\big) & \wedge & \big(d \neq \bot \Rightarrow M(d)\big) \end{array}\right). \tag{5.2}$$

5.3.2 The verification approach

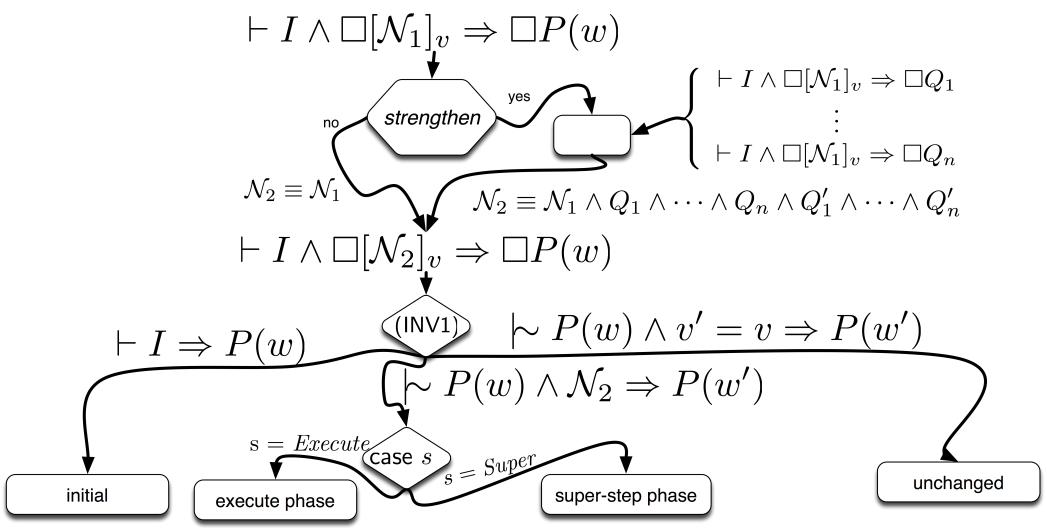


Figure 5.3: Proof plan of a Hume invariant

Figure 5.3 shows the overall proof plan of a Hume invariant. Here, the hexagon denotes OR choice, diamond denotes AND choice, and a box is a proof technique (e.g. a tactic). Now, with the exception of the last case-analysis on the scheduler s, this is a similar structure, as one would expect to that of a generic TLA invariant. First, $\mathcal{N}$ is strengthened by the (already verified) invariants $Q_1, \ldots, Q_n$ in both the before (Q_i) and after (Q'_i) states. The resulting conjecture is then unlifted from temporal- to the state- and action-levels, using the standard TLA rule (INV1). This creates three sub-goals: the first is the initial state $\vdash I \Rightarrow P(w)$, which is assumed to be trivially handled by the initial technique; the last sub-goal is the unchanged sub-script $\mid\!\sim P(w) \wedge v' = v \Rightarrow P(w')$, handled by the trivial unchanged technique; the second

sub-goal is the next action step $\mid\!\sim P(w) \wedge \mathcal{N}_2 \Rightarrow P(w')$. The proof is by a case-split on the scheduler s, where each of the resulting phases (sub-goals) are discussed separately below.

The execute phase

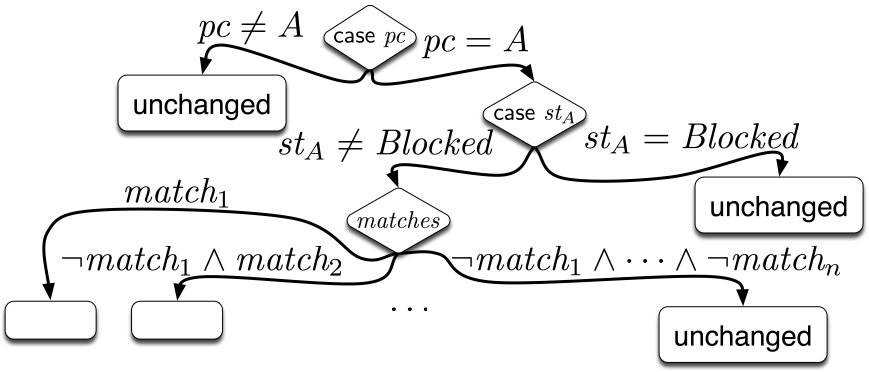


Figure 5.4: Proof plan of execute phase

The proof plan of the execute phase is shown in Figure 5.4. If the invariant is over wires, then the wires are either left unchanged or consumed. Thus, although each case of Figure 5.4 is trivial, this structure is still required for wires, unless a meta-theorem has already been verified capturing that wires are either left unchanged or consumed, which cannot be expressed in a shallow embedding. The first step is a case-split on $pc = A$. Thus, the focus is on invariants updated by one box, i.e. not (D4) style invariants. For such invariants, all combinations of enumerated cases (see Figure 5.4) must be combined. This will still work, but generates several unnecessary sub-goals, and a full case-analysis on pc may be better.

Now, $pc \neq A$ is trivially handled by unchanged. The $pc = A$ case follows by a case split on the *Blocked* state of A. The *Blocked* case is trivially handled by unchanged, while the non-*Blocked* case induces case-analysis on each match, in addition to the *Matchfail* case, which is trivially handled by unchanged. The matches depend on the property and box, and a generic structure cannot be created.

The super-step phase

Figure 5.5 shows the proof plan of the super-step phase of a single wire/result buffer projection invariant. Both cases start with a case-split on output assertion *ao*. This is sufficient for the result buffer invariant, as shown in the figure. For a wire invariant, the *ao* case follows by case-split on the possible return value of *nw*.

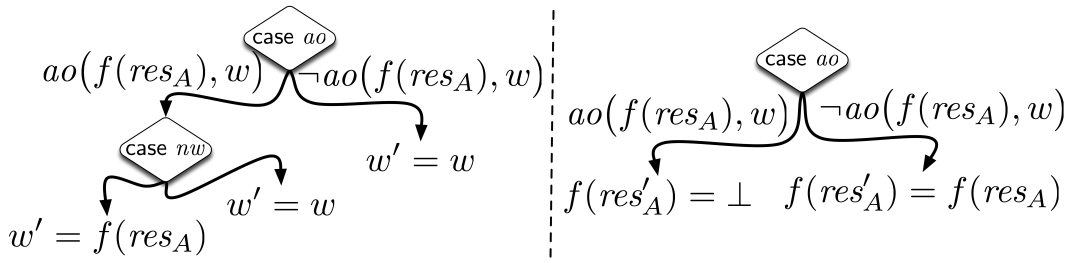


Figure 5.5: Proof plan of super-step phase. Left: proof of wire invariant. Right: proof of result buffer invariant.

The case where the invariant is over more than one wire has previously been discussed in Section 5.2. For a result buffer this case is handled in the same way as a single projection.

5.3.3 Isabelle/Hume tactics developed

The following tactics implement the reasoning techniques described above. As in the previously defined tactics, the rewriting is performed by the *simplifier*. Note that a tactic may work on several sub-goals, and reasoning on one sub-goal may solve it or create new sub-goals. To ensure that the correct technique is applied to the correct sub-goal, the techniques are applied backwards with respect to the sub-goals' number (last sub-goal is reasoned with first):

inv_strengthen_tac rews invs goal implements the invariant strengthening mechanism shown in Figure 5.3. It requires that the given goal sub-goal is of the form $\vdash I \wedge \Box[\mathcal{N}]_v \Rightarrow F$, while invs is a list of theorems of the same form where F has to be of the form $\Box G$. Now, rews is a list of rewrites which are applied to both goal and all the theorem in invs. This mainly involves unfolding the name of the program specification. All the invariants are then joined into a conjunction of theorems. This is achieved by applying the rule (see Section 3.6):

theorem `inv_join`: **assumes:** $\vdash$ P $\longrightarrow \Box$Q **and** $\vdash$ P $\longrightarrow \Box$R
shows: $\vdash$ P $\longrightarrow \Box$(Q $\wedge$ R),

recursively for each adjacent element in the invs list. This creates a "conjunction invariant" applied to goal using the rule (also in Section 3.6):

theorem `spec_inv2_mono`:
assumes: $\vdash$ I $\wedge \Box$[N]_v $\longrightarrow \Box$J **and** $\vdash$ I $\wedge \Box$[N $\wedge$ J $\wedge \circ$J]_v $\longrightarrow$ P
shows: $\vdash$ I $\wedge \Box$[N]_v $\longrightarrow$ P.

This, with the "conjunction invariant" reduces the goal to the second assumption.

execute_tac rews goal implements the execute phase described above. rews must at least contain the definition of the execute phase of the box(es) updating the wire/result buffer of the invariant. This is first used to simplify the goal. As shown in Chapter 4, the execute phase of a box action, is represented by a case-split on *pc*, followed by a case-split on the box's *Blocked* state, both represented as if-then-else expressions in Isabelle/HOL. The matches are also represented by nested if-then-else expression. The structure of the execute phase is thus implemented by splitting each of these if-then-else expressions into two sub-goals. If there is more than one box, all possible combinations are enumerated. The standard simplifier, which contains the assumptions (the match and strengthened invariants), and the simplify_hume_tac and unchanged_tac are then used to attempt to solve all the sub-subgoals.

superstep_tac sup goal mechanises the super-step phase described above. For each box, the super-step action contains an if-then-else expression, where the condition holds if the outputs are asserted. Depending on the context of the invariant and program, there are many ways to deal with the super-step. This is handled by introducing an inductive data type supType. sup must be of this type, and it contains the following constructors: SupNothing leaves the goal unchanged. (SupGen rews) requires that rews must at least contain the definition of the execute phase of the box(es) updating the wire/result buffer of the invariant. This is so for all remaining cases. rews is used to unfold the required definitions, followed by a case-split on the *ao* if-then-else expression, and application of nwireE on the *ao* case, as described in Section 5.2. If there is one box with n output wires, the application of nwireE will create $2^n + 1$ sub-goals, an example of which is given in the left tree of Figure 5.5. If there is more than one box, the number of sub-goals changes such that all possible combinations are enumerated. (SupRes rews) relates to the right tree of Figure 5.5, and is used if the invariant is on the result buffer. After the unfolding, the *ao* if-then-else expression is split into two sub-goals. (SupCase (rews,cs)) unfolds and splits the *ao* if-then-else expression. In the *ao* case, it then applies (recursively) case-splits on all the elements in the cs list, followed by rewriting using nw1 and nw2, as described in Section 5.2. The cs list is assumed to be of the form isVal<$\cdots$>. To achieve the "many wires predicate" case, also described in Section 5.2, the theorem must first be strengthened by all the necessary isVal<A>=isVal<B> lemmas; finally, (SupComp (rews,cs)) first applies (SupCase (rews,cs)), followed by nwireE on the remaining nwires.

humeinv_tac invs rews sup exe init attempts to solve a Hume invariant. It first sim-

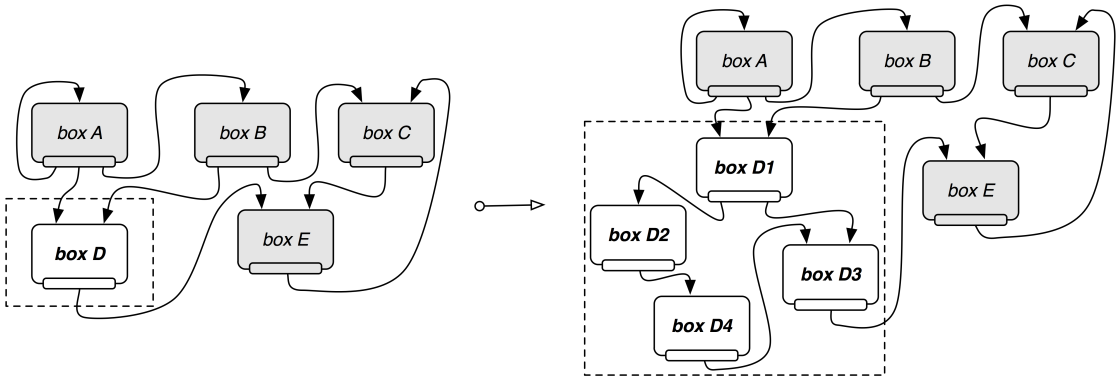


Figure 5.6: Transformation illustration

plifies using rews, followed by invariant strengthening using inv_strengthen_tac rews invs 1. The following rule (see Section 3.6) unlifts the goal to the action level:

theorem `invmono`: **assumes:** $\vdash I \longrightarrow P$ **and** $\vdash\!\!\sim P \wedge [N]\_f \longrightarrow \circ P$
shows: $\vdash I \wedge \Box[N]\_f \longrightarrow \Box P$

The first goal is unlifted to the Isabelle/HOL level, and unl_tac and the init rewrite rules are applied to solve the goal. The rule (see Section 3.6)

theorem `preimpsplit`:
assumes: $\vdash\!\!\sim I \wedge N \longrightarrow Q$ **and** $\vdash\!\!\sim I \wedge \mathsf{Unchanged}\ v \longrightarrow Q$
shows: $\vdash\!\!\sim I \wedge [N]\_v \longrightarrow Q$

is applied to the second subgoal. The second is applied to unchanged_tac, followed by unlifting to Isabelle/HOL, and applying the standard simplifier. The first subgoal is unlifted to Isabelle/HOL, followed by case-split on $w \vDash \$s$, creating the $(w \vDash \$s) = \mathsf{Execute}$ and $(w \vDash \$s) = \mathsf{Super}$ cases. execute_tac exe 1 is applied to the first, while superstep_tac sup 2 is applied to the second.

The tactics are used in the case-studies in Sections 5.5.2 and 5.5.3. They are also used in the tactics developed in Chapter 6.

5.4 Program transformation verification[1]

Transformations in Hume are mainly motivated by a failed costing in the expression layer, as shown by the Hume methodology described in Section 2.6. Since boxes are executed in separation, and the costing will fail in the (expressive) expression layer, the transformation will involve moving computation from the expression layer of one

[1]Parts of the motivations discussed in this section has been published in [89].

box, into the coordination layer. Thus, one box is split up into many boxes, or extra wires are introduced. In both cases the source of the transformation is always one box. This is illustrated in Figure 5.6 where box `D` is transformed into the spatial component of boxes `D1` to `D4`.

The state components, that is connected wires, result buffer and state variable, of the source program must be "reused" in the transformed program. The unchanged boxes `A`,`B`,`C` and `E` of Figure 5.6, reuse their state components. The input wires of `D` become the input wires of `D1`, and the output wires of `D` become the output wires of `D4`. Thus, it must be the case that $res_{\texttt{D}}$ becomes $res_{\texttt{D4}}$. In the source program the input wires are consumed and $res_{\texttt{D}}$ are updated in the same step, while in the resulting program $res_{\texttt{D4}}$ is updated several cycles after `D1` consumes the inputs. Since wires and result buffers are free, that is, not $\exists$-bound, in the specification, a refinement mapping which can capture this delay does not exist. Thus, a one-step box execution is too abstract to reason about transformations. To verify a transformation, the execute step of a box $b \in BS$ must split into two distinct steps: the pattern matching and *consume* step is split from the *compute* step, which updates the result buffer res_b. This requires introducing an input buffer inp_b for all boxes $b \in BS$, which contains the consumed value and is used in the compute step of the same box. This can be embedded by replacing BS with the $\{b^{con}.\ b \in BS\} \cup \{b^{com}.\ b \in BS\}$, such that $pc = b^{con}$ executes the consume step of box b and $pc = b^{com}$ executes the compute step, and $b^{con} \prec b^{com}$. However, since boxes are executed sequentially, it is sufficient to introduce a boolean variable *con*, where *con* denotes the consume step of the current box, and $\neg con$ denotes the compute step. This approach is formalised below.

5.4.1 A two-step box execution formalised

To split the execute step from Section 4.3.3, a new function,

$$
\begin{array}{l}
pattcopy(patt, iws) \triangleq \\
\quad \textbf{case} \quad iws = \langle\rangle \qquad\qquad\qquad\quad \rightarrow \langle\rangle \\
\quad\ \ \square \quad iws \neq \langle\rangle \wedge hd(patt) = * \ \rightarrow \langle\bot\rangle \cdot pattcopy\big(tl(patt), tl(iws)\big) \\
\quad\ \ \square \quad iws \neq \langle\rangle \wedge hd(patt) \neq * \ \rightarrow \langle hd(iws)\rangle \cdot pattcopy\big(tl(patt), tl(iws)\big),
\end{array}
$$

is required. Based on a given pattern *patt*, a tuple holding the input wires *iws* that are consumed by the *consume* function, is returned. This is used to update the box input

buffer *inp* in the function:

$$
\begin{array}{l}
execute^{con}(rs, iws, expr) \triangleq \\
\quad \textbf{case} \quad rs = \langle\rangle \qquad\qquad\qquad\qquad \rightarrow \langle iws, \langle \bot, \cdots \bot \rangle, expr, Matchfail \rangle \\
\quad\quad \square \quad rs \neq \langle\rangle \wedge pm\big(patt\big(hd(rs)\big), hd(iws)\big) \quad \rightarrow \left\langle \begin{array}{l} consume\big(patt\big(hd(rs)\big), iws\big), \\ pattcopy\big(patt\big(hd(rs)\big), iws\big), \\ exp\big(hd(rs)\big), Runnable \end{array} \right\rangle \\
\quad\quad \square \quad rs \neq \langle\rangle \wedge \neg pm\big(patt\big(hd(rs)\big), hd(iws)\big) \quad \rightarrow execute^{con}(tl(rs), iws, expr).
\end{array}
$$

Based on the pattern matching, the correct expression must be executed in the compute step. The *expr* variable stores this expression. Now, given a box rule set rs; input wires iws; and $expr$, $execute^{con}$ return a quadruple of: the consumed input wires; the new input buffer; the new $expr$; and the new box state.

$\mathcal{B}_i^{e^2}$ updates $\mathcal{B}_i^e$ for the two-step box execution. The $st_i \neq Blocked$ guard is now split into two cases: in the consume step (con) the input wires are consumed, and copied to the input buffer. In the compute step ($\neg con$) the expression is executed:

$$
\begin{array}{rlcl}
\mathcal{B}_i^{e^2} \triangleq & st_i \neq Blocked \wedge con & \Rightarrow & \langle iws_i, inp_i, expr, st_i \rangle' = execute^{con}(rs_i, iws_i, expr) \\
 & & \wedge & res_i' = res_i \\
\wedge & st_i \neq Blocked \wedge \neg con & \Rightarrow & \langle expr, iws_i, inp_i, res_i, st_i \rangle' = \\
 & & & \quad \langle expr, iws_i, inp_i, run(expr, inp_i), Runnable \rangle \\
\wedge & st_i = Blocked & \Rightarrow & \langle expr, iws_i, inp_i, res_i, st_i \rangle' = \langle expr, iws_i, inp_i, res_i, st_i \rangle.
\end{array}
$$

$\mathcal{B}_i^{s^2}$ updates the super-step action $\mathcal{B}_i^s$ with the input buffer:

$$
\begin{array}{rl}
\mathcal{B}_i^{s^2} \triangleq & \big(ao(res_i, ows_i) \Rightarrow \\
 & \quad \langle ows_i, inp_i, res_i, st_i \rangle' = \langle nw(res_i, ows_i), \langle \bot, \cdots \bot \rangle, \langle \bot, \cdots \bot \rangle, Runnable \rangle\big) \\
\wedge & \big(\neg ao(res_i, ows_i) \Rightarrow \\
 & \quad \langle ows_i, inp_i, res_i, st_i \rangle' = \langle ows_i, inp_i, res_i, Blocked \rangle\big)
\end{array}
$$

$\mathcal{S}_{seq}^2$ updates $\mathcal{S}_{seq}$ with con which ensures that the compute step of *last_box* is also executed:

$$
\begin{array}{rlcl}
\mathcal{S}_{seq}^2 \triangleq & (s = Execute \wedge (pc' \neq last\_box \vee \neg con') & \Rightarrow & s' = Execute) \\
\wedge & (s = Execute \wedge (pc' = last\_box \wedge \quad con') & \Rightarrow & s' = Super) \\
\wedge & (s = Super & \Rightarrow & s' = Execute).
\end{array}
$$

$\mathcal{N}_{seq}^2$ updates $\mathcal{N}_{seq}$ in the following way: the input buffer inp_i is added; con is updated; $expr$ is updated in the super-step; pc is only updated when con' hold. Note that a box is only executed in the compute step if it is *Runnable*: a *Blocked* or *Matchfail* box is

left unchanged, and the environment is one-step:

$$\begin{array}{lll}
\mathcal{N}^2_{seq} \triangleq & \big(s = \mathit{Execute} & \Rightarrow (\mathit{pc} \in \mathit{BS} \Rightarrow \mathcal{B}^{e^2}_{\mathit{pc}}) \\
& & \wedge \big(\bigwedge_{i \in \mathit{BS} - \{\mathit{pc}\}} \langle \mathit{iws}_i, \mathit{inp}_i, \mathit{res}_i, \mathit{st}_i \rangle' = \langle \mathit{iws}_i, \mathit{inp}_i, \mathit{res}_i, \mathit{st}_i \rangle \big) \\
& & \wedge\ \mathit{con}' = \textbf{if } \mathit{pc} = \mathit{env} \vee \mathit{st}'_{\mathit{pc}} \neq \mathit{Runnable} \textbf{ then } \mathit{True} \textbf{ else } \neg \mathit{con} \\
& & \wedge\ \mathit{pc}' = \textbf{if } \mathit{pc} \neq \mathit{env} \wedge \mathit{st}'_{\mathit{pc}} = \mathit{Runnable} \Rightarrow \neg \mathit{con} \\
& & \qquad \textbf{then } \mathit{next_box(pc)} \textbf{ else } \mathit{pc}\big) \\
\wedge & (s = \mathit{Super} & \Rightarrow (\bigwedge_{i \in \mathit{BS}} \mathcal{B}^{s^2}_i) \wedge \mathit{pc}' = \mathit{first_box} \wedge \mathit{con}' = \mathit{True} \wedge \mathit{expr}' = \mathit{expr}\big).
\end{array}$$

The initial state and full program specification is then updated as follows:

$$\begin{array}{lll}
I^2 & \triangleq & I \wedge (\mathit{pc} = \mathit{first_box}) \wedge (\mathit{con} = \mathit{True}) \wedge (\mathit{expr} = \langle \bot, \cdots, \bot \rangle) \\
& \wedge & \bigwedge_{i \in \mathit{BS}} \mathit{inp}_i = \langle \bot, \cdots . \bot \rangle \\
H^l_{seq^2} & \triangleq & I^2 \wedge \Box[\mathcal{N}^2_{seq} \wedge \mathcal{S}^2_{seq} \wedge \mathcal{E}]_{\langle s, ws, inp, res, st, pc, con, expr \rangle} \\
H_{seq^2} & \triangleq & \exists\ \mathit{st}, s, \mathit{pc}, \mathit{con}, \mathit{expr}.\ H^l_{seq^2}.
\end{array}$$

A comparison with one-step box execution

An invariant proof now requires an additional case-split on *con*, while a wire invariant proof may require a lemma on the matching in addition to a lemma updating the result buffer. One- and two-step box execution embedding can be combined, such that only transformed boxes are given a two-step representation: let $\mathit{BS}_{\mathit{2\text{-}step}} \subseteq \mathit{BS}$ contain all the boxes that should be executed in two-steps. The two representations can be combined by replacing $(\mathit{pc} \in \mathit{BS} \Rightarrow \mathcal{B}^{e^2}_{\mathit{pc}})$ in the first line of $\mathcal{N}^2_{seq}$ by $\big(\mathit{pc} \in \mathit{BS} \Rightarrow (\mathit{pc} \in \mathit{BS}_{\mathit{2\text{-}step}} \Rightarrow \mathcal{B}^{e^2}_{\mathit{pc}}) \wedge (\mathit{pc} \notin \mathit{BS}_{\mathit{2\text{-}step}} \Rightarrow \mathcal{B}^{e}_{\mathit{pc}})\big)$ and $\mathit{pc} = \mathit{env} \vee \mathit{st}'_{\mathit{pc}} \neq \mathit{Runnable}$ in the third line of the same action with $\mathit{pc} \notin \mathit{BS}_{\mathit{2\text{-}step}} \vee \mathit{pc} = \mathit{env} \vee \mathit{st}'_{\mathit{pc}} \neq \mathit{Runnable}$.

Finally, since *res* and *ws* are free, it is not the case that the two-step execution refines the one-step execution. The reason for this was discussed in great detail in Section 4.2.

5.4.2 The verification of a transformation

In TLA, Hume transformations are represented as *refinements*: a TLA embedded Hume program H_1 is transformed into a program H_2, written $\mathit{Trans}(H_1, H_2)$, iff H_2 *refines* H_1. Here, refinement is the same as implementation, and implementation is represented as implication. Thus, this is defined as:

$$\mathit{Trans}(H_1, H_2) \ \equiv\ H_2 \Rightarrow H_1. \tag{5.3}$$

Let I_1 and I_2 be the initial states, and $\mathcal{N}_1$ and $\mathcal{N}_2$ the next actions, and w and v the state space, including hidden variables, of the two respective specifications. This can then be written as

$$(\exists\, st, s, pc, con, expr.\ I_2 \wedge \Box[\mathcal{N}_2]_w) \Rightarrow (\exists\, st, s, pc, con, expr.\ I_1 \wedge \Box[\mathcal{N}_1]_v).$$

To verify this conjecture, rule (E2) is first applied to all ∃-bound variables of the implication premise (H_2), which removes the quantifiers, and replaces the bound variables by new constants. Next, the refinement mapping for the ∃-bound variables in H_1 must be found. Let $\overline{F}$ be F with all ∃-bound variables replaced by the refinement mapping: $F[\overline{st}/st, \overline{s}/s, \overline{pc}/pc, \overline{con}/con, \overline{expr}/expr]$. This substitution is achieved by rule (E1). Consequently, all ∃ quantifiers in the implication conclusion are removed, and the bound variables are replaced by the witnesses defined by the refinement mapping. Then, by applying (TLA2) and (STL4) the temporal formula is unlifted to the action level, by verifying the initial state and the step cases separately. This is summarised in the following proof-tree which should be read backwards:

$$\dfrac{\dfrac{\dfrac{\dfrac{w' = w \Rightarrow \overline{v}' = \overline{v} \qquad \mathcal{N}_2 \Rightarrow [\overline{\mathcal{N}_1}]_{\overline{v}}}{[\mathcal{N}_2]_w \Rightarrow [\overline{\mathcal{N}_1}]_{\overline{v}}}\,([\cdots]_{\ldots}) \qquad I_2 \Rightarrow \overline{I_1}}{(I_2 \wedge \Box[\mathcal{N}_2]_w) \Rightarrow (\overline{I_1} \wedge\ \Box[\overline{\mathcal{N}_1}]_{\overline{v}})}\,((\mathsf{TLA2})\text{ using }(\mathsf{STL4}))}{(I_2 \wedge \Box[\mathcal{N}_2]_w) \Rightarrow (\exists\, st, s, pc, con, expr.\ I_1 \wedge \Box[\mathcal{N}_1]_v)}\,(\mathsf{E1})}{(\exists\, st, s, pc, con, expr.\ I_2 \wedge \Box[\mathcal{N}_2]_w) \Rightarrow (\exists\, st, s, pc, con, expr.\ I_1 \wedge \Box[\mathcal{N}_1]_v)}\,(\mathsf{E2})$$

Note that to allow internal steps in the more concrete $\mathcal{N}_2$ action, $\mathcal{N}_2 \Rightarrow [\overline{\mathcal{N}_1}]_{\overline{v}}$ is verified instead of $\mathcal{N}_2 \Rightarrow \overline{\mathcal{N}_1}$. The main work in the proof is this sub-goal. Section 5.5.4 discusses transformation verification by example, and for reasons shown there, a generic structure proof structure of $\mathcal{N}_2 \Rightarrow [\overline{\mathcal{N}_1}]_{\overline{v}}$ cannot be given.

5.5 Hume verification case studies

In this section the embeddings and reasoning techniques outlined above and in the previous chapter are applied to four case studies: Section 5.5.1 discusses invariant verification by model checking in TLC; Section 5.5.2 and Section 5.5.3 discuss invariant verification in Isabelle/Hume; while Section 5.5.4 discusses transformations. Section 5.5.3 and Section 5.5.4 illustrate the problem with reasoning in the Hume coordination layer, and are used as motivations for *Hierarchical Hume* in Chapter 6. Several of the tactics developed above are also used in case-studies in Chapter 7.

5.5.1 Case study H1: traffic lights in TLC[2]

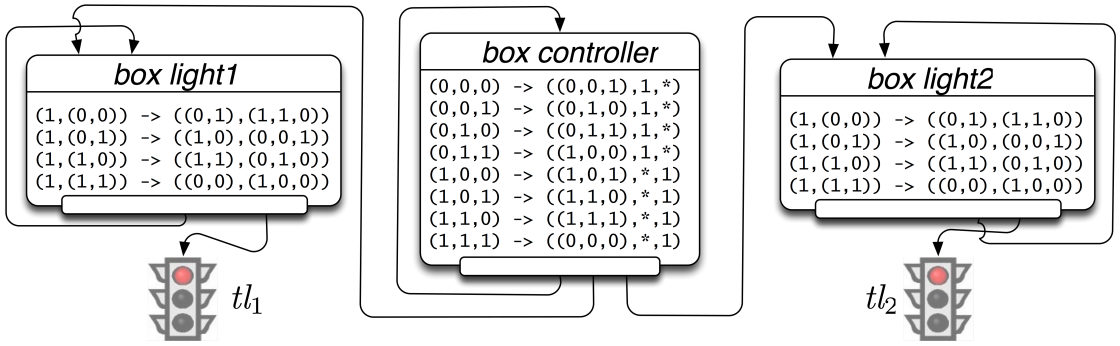


Figure 5.7: A traffic light system in Hume.

The use of TLC, instead of Isabelle/Hume, liberates the user from the proofs, and is ideal to test the embedding and experimental case studies, like the real time properties shown below. Now, in this section, the example shown in Figure 5.7, is discussed. It is embedded as discussed in Section 4.4. The program consists of two light boxes, `light1` and `light2`, and a controller box `controller`. The program is in HW-Hume, which only supports `0`s,`1`s and tuples of `0`s and `1`s. The light boxes are in charge of switching the correct colours on their associated traffic light. This is achieved by sending a triple `(X,Y,Z)` to the light, where a 1 switches the light on and 0 off. Furthermore, `X` refers to the top red light, `Y` the middle amber light, and `Z` the bottom green light. The light box is in charge of changing the lights in the correct order, as used in the UK: red is followed by red and amber, which is followed by green, then amber, and finally, red again. The `controller` box is responsible for co-ordinating the two lights. It is not required to know which lights are on. All it knows is that a sequence from red, back to red requires 4 steps. It sends a signal to a light box when it should change colour. Thus, 4 sequential signals are sent to `light1`, then 4 sequential signals are sent to `light2`, then to `light1` again, and so on. To achieve this, each box needs to know their state, represented by feedback wires. The lights are assumed to initially be red. Let

$$\mathit{Init} \wedge \Box[\mathcal{N} \wedge \mathcal{S} \wedge \mathcal{E}]_v. \tag{5.4}$$

be the embedding of the program in TLA$^+$, where v is the state-space of the program. Note that since the program is in HW-Hume, *Nat* is replaced by $\{0, 1\}$ in the configuration file. Instead of reasoning directly with wires, i.e. of the form $\Box\big(w \neq \bot \Rightarrow P(w)\big)$, the actual physical lights are embedded as variables tl_1 and tl_2. These are seen as part

[2]The example in this section has been published in [91].

of the *system* (not program), and not ∃-bound. The action $\mathcal{N}_{tl_1,tl_2}$ updates these variables if (and only if) the corresponding light changes colour – and $Init_{tl_1,tl_2}$ sets them to be red initially. Note that these auxiliary definitions do not change the behaviour of (5.4):

$$Init \wedge Init_{tl_1,tl_2} \wedge \Box[\mathcal{N} \wedge \mathcal{S} \wedge \mathcal{E} \wedge \mathcal{N}_{tl_1,tl_2}]_{\langle tl_1,tl_2,v\rangle}. \tag{5.5}$$

(5.5) is used for the verification. (5.6) is an invariant asserting that both lights cannot be green at the same time:

$$\Box\big(tl_1 \neq (0,0,1) \vee tl_2 \neq (0,0,1)\big) \tag{5.6}$$

However, (5.5) is stronger than that. (5.6) can be strengthened to show that if one of the lights is not red, then the other light is red:

$$\Box\Big(\big(tl_1 \neq (1,0,0) \Rightarrow tl_2 = (1,0,0)\big) \wedge \big(tl_2 \neq (1,0,0) \Rightarrow tl_1 = (1,0,0)\big)\Big)$$

TLA can also be used to reason about computation. This requires a before and after state, and must thus be represented as an action. For example, an action can express that the *order* in which the lights change is correct. Given a current light, the (Hume) function `Next` returns the correct next value in the UK sequence:

```
Next l = case l of (1,0,0) -> (1,1,0) | (1,1,0) -> (0,0,1)
                 | (0,0,1) -> (0,1,0) | (0,1,0) -> (1,0,0);
```

Let *Next* be a direct porting of the Hume `Next` function into a TLA$^+$ operator. The properties have to be defined separately for each of the two lights:

$$\Box[tl_1' = Next\,(tl_1)]_{tl_1} \qquad\qquad \Box[tl_2' = Next\,(tl_2)]_{tl_2}$$

The super-scripts tl_1 and tl_2 ensure that only those steps where the lights actually change value are considered, which the validity depends on.

Real-time Hume model checking

The final example shows how TLC can be used to verify real-time properties of the coordination layer, using a technique known as *explicit-time* model checking [125]. The focus is on upper time bounds, i.e. properties of the form

"*if P holds then Q will always hold within T time units.*"

To prove this, the costs for the expression layer are required, which can be found by applying the analysis described in e.g. [29]. Moreover, the cost of copying in the coordination layer must also be obtained. The exact property specifies that if the first light is green then it will become red within 100 ms

"*if* tl_1 = `(0,0,1)` *holds then* tl_1 = `(1,0,0)` *will always hold within* 100 *ms.*"

To achieve this, (5.5) is augmented with a counter t, with the following type invariant:

$$\Box\big(t \in (\mathit{Int} \cup \{\mathit{Error}, \mathit{Disabled}\})\big).$$

Disabled means that the first property (tl_1 = `(0,0,1)`) has not occurred yet, or the second property (tl_1 = `(1,0,0)`) has occurred within the time bound; *Error* indicates that the bound (`100` ms) has been reached before the second property holds; while $t \in \mathit{Int}$ indicates that the first property has occurred and the second property and the time bound has not been reached.

Let NT be an operator which takes the counter and returns the new value, based on which boxes executes and the scheduling. The real-time specification with the counter t is then defined as:

$$\mathit{Init} \wedge \mathit{Init}_{tl_1,tl_2} \wedge t = \mathit{Disabled} \wedge \Box[\mathcal{N} \wedge \mathcal{S} \wedge \mathcal{E} \wedge \mathcal{N}_{tl_1,tl_2} \wedge t' = \mathit{NT}(t)]_{\langle t,tl_1,tl_2,v\rangle}.$$

Let T_{l1}, T_{l2} and T_{ctl} be the upper time bound for the execute phase when the two lights and `controller` box execute, and T_{super} the upper-bound of the super-step. The cost of copy, scheduling and so on of the coordination layer must be found. However, these values are here assumed to be $T_{l1} = 2, T_{l2} = 2, T_{ctl} = 1, T_{super} = 1$. The environment has no cost, although this assumption may be false. Regardless, the approach is the same.

t is initially *Disabled*. For each step *NT(t)* calculates the new value of t as follows: if $tl_1' = (1, 0, 0)$ then t is set to *Bound*. For all the following steps t is decremented with either T_{l1}, T_{l2}, T_{ctl} or T_{super}, depending on the scheduling phase, and which box is executed in the *Execute* phase. If $tl_2' = (0, 0, 1)$ then t is reset to *Disabled*. Finally, if $t \leq 0$ then t' is set to *Error*. The desired property is then formalised as:

$$\Box(t \neq \mathit{Error})$$

In this example, the lowest guarantee with these assumptions was 60 ms. Note that [125] suggests that using TLC for explicit time model checking may not be too inefficient compared to more specialised real time model checkers.

$$(\mathsf{E0})\ 0 \in \mathsf{EvenN}$$

$$(\mathsf{ES})\ \frac{\mathsf{X} \in \mathsf{OddN}}{\mathsf{Suc\ X} \in \mathsf{EvenN}} \qquad (\mathsf{OS})\ \frac{\mathsf{X} \in \mathsf{EvenN}}{\mathsf{Suc\ X} \in \mathsf{OddN}}$$

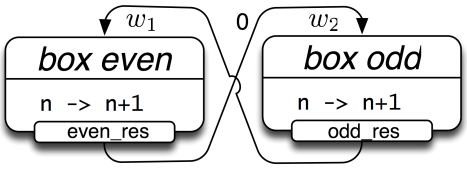


Figure 5.8: Left: mutually inductive defined EvenN *and* OddN *sets in Isabelle/HOL. Right: Hume program with* `even` *and* `odd` *boxes.*

5.5.2 Case study H2: even and odd numbers

The first Isabelle/Hume case study is the program shown on the right hand side of Figure 5.9. It consists of two boxes connected by two wires, and forming a closed system. The property that should be verified is that wire w_1 is always an odd number (or empty), and wire w_2 is always an even number (or empty). These properties are specified using two sets EvenN and OddN, defined in Isabelle/HOL by mutual induction. The required introduction rules are shown on the left side of Figure 5.9. These properties are formalised as follows:

$$\Box(w_1 \neq \bot \Rightarrow w_1 \in \mathsf{OddN}) \tag{5.7}$$

$$\Box(w_2 \neq \bot \Rightarrow w_2 \in \mathsf{EvenN}) \tag{5.8}$$

Proof outline. Both these properties are (D4) style invariants with mutual dependency. Thus, the following theorem must first be verified

$$\Box \left(\begin{array}{ccccc} & (\mathsf{odd\_res} \neq \bot \Rightarrow \mathsf{odd\_res} \in \mathsf{OddN}) & \wedge & (w_1 \neq \bot \Rightarrow w_1 \in \mathsf{OddN}) \\ \wedge & (\mathsf{even\_res} \neq \bot \Rightarrow \mathsf{even\_res} \in \mathsf{EvenN}) & \wedge & (w_2 \neq \bot \Rightarrow w_2 \in \mathsf{EvenN}) \end{array} \right)$$

The mechanised Isabelle/Hume version, called mainth, is listed in Appendix A.3.1. There, (5.7) is called oddwire, while (5.8) is called evenwire. mainth is proved by the humeinv_tac followed by the application of the definitional rules of the EvenN and OddN shown in Figure 5.9. (5.7) and (5.8) follow by the application of (STL5) to mainth. ∴

5.5.3 Case study H3: a vending machine

Figure 5.9 shows a vending machine in Hume, and is an adaptation of an example in [90]: the `inp` box accepts coins and button input from the hardware; these are sent to the `control` box, which handles the request. The current balance is kept in the w_4 feedback loop, and an infinite amount of tea and coffe is assumed; the result of this is sent to `outp` which either returns nothing, or tells the machine to produce coffee (`0` is

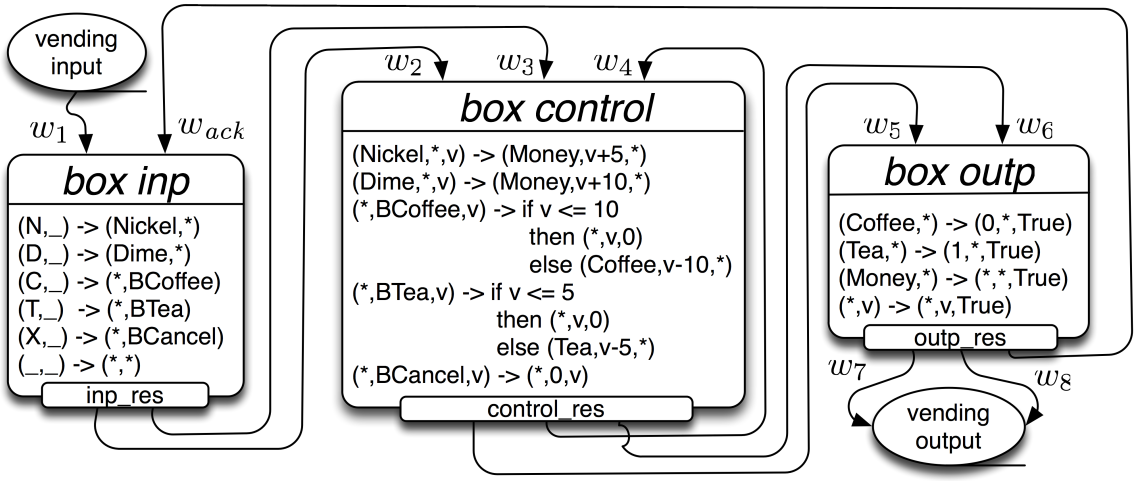


Figure 5.9: Vending machine in Hume.

sent to wire w_7), tea (`1` is sent to wire w_7), or the balance is returned on wire w_8. This requires the following types:

```
data coins   = Nickel | Dime;
data drinks  = Coffee | Tea | Money;
data buttons = BCoffe | BTea | BCancel;
data input   = N | D | C | T | X;
```

The first property asserts that the balance is never negative:

$$\Box(w_4 \neq \bot \Rightarrow w_4 \geq 0) \tag{5.9}$$

Proof outline. This is a (D2) type invariant, which first requires the proof of

$$\Box\big((w_4 \neq \bot \Rightarrow w_4 \geq 0) \wedge \mathsf{control\_res}_2 \neq \bot \Rightarrow \mathsf{control\_res}_2 \geq 0\big).$$

This is first automatically verified by humeinv_tac, and (5.9) follows by rule (STL5). In Appendix A.3.2 this is mechanised as vend_mon1, while (5.9) is mechanised as vend_money. ∴

These three boxes create a spatial component, with several partial correctness properties: 'if coins are deposited, then the balance is increased accordingly'; 'if coffee is requested, and there are sufficient funds, then coffee is produced'; and the opposite: 'if coffee is requested, and there are not enough funds, then coffee is not returned'.

The input of this component is w_1, while the result is produced on w_7 or w_8, meaning it will take three cycles to produce a result. Moreover, the w_{ack} wire has been added to the original version [90], where a new computation could be started before the

previous had terminated. Specifying partial correctness properties requires introducing auxiliary variables, and specifying the properties using them. However, since they are not part of the program, these properties are not preserved by transformations. These variables must remember the input when `inp` first executes (w_{ack} is consumed) until an output is produced (w_{ack} is written to).

To achieve such specification, it must be shown that during computation (w_{ack} is empty), the auxiliary variables are unchanged. Moreover, in order to verify partial correctness it must be shown that execution is sequential. This requires to show that, with the exception of w_4, w_7 and w_8, only one of the remaining wires or result buffers is non-empty. Using this, it can be shown that the auxiliary variables are unchanged during execution of the component, and can thus be used to specify partial correctness with the output buffer of `outp`. Verifying, or even using this property, causes the simplifier to loop, meaning that the humeinv_tac fails. Thus, it has not been proved.

Irrespective of the failure of the humeinv_tac tactic, this example has illustrated that partial correctness properties over a spatial component cannot be specified naturally. Even if the component is restricted, for example by disabling pipe-lining, specification becomes difficult and requires the introduction of auxiliary variables. Consequently, such properties will not be preserved by transformations, which is a key feature in Hume development. The next chapter introduces an extension to Hume, which allows such properties to be captured naturally, and enables automation of such proofs.

5.5.4 Case study H4: adder transformations verification[3]

A first attempt

This section discusses verification of the transformation shown in Figure 5.10. Here, a full ripple carry adder represented as a truth table (the `adder` box) is transformed into a spatial component consisting of two half adders (the `half1` and `half2` boxes) and a logical OR gate (box `or`). The dotted wires are introduced by the transformation. The verification is by model checking in TLC, where the embedding described in Section 4.4 is updated with two-step box execution. Let P_1 be the TLA embedding of the source program, and P_2 the result. The first non-trivial step is to find the refinement mapping for the $\exists$-bound variables. The $s = \mathit{Execute}$ and $s = \mathit{Super}$ steps are discussed separately.

When $s = \mathit{Execute}$, the $pc =$ `gen` and $pc =$ `show` cases are trivial since they are unchanged by the transformation. `half1`, `half2` and `or` make up the behaviour of `adder`. For both *con* and $\neg$*con*, $\overline{pc}$ must be one of these boxes since the result buffers

[3]Parts of this section has been published in [89].

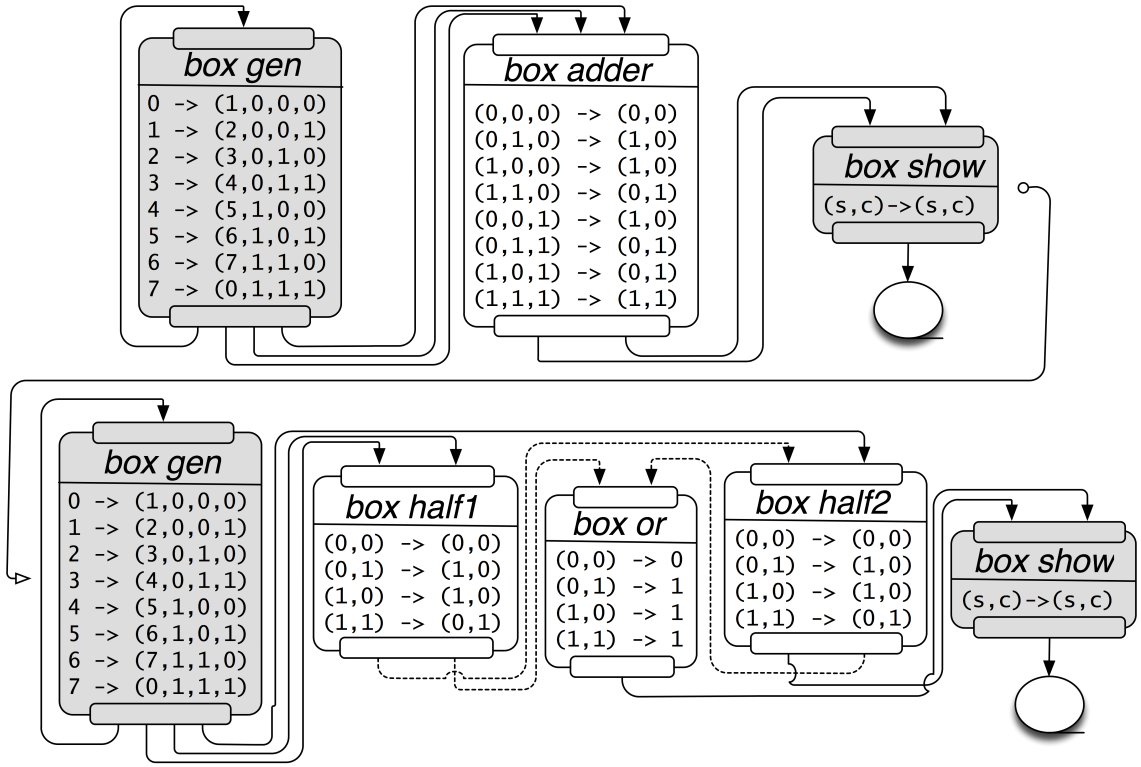


Figure 5.10: Hume program transformation example.

and wires are free. However, `half1` and `half2` together consume the wires as `adder`, and in different cycles. Moreover, `half2` and `or` produces the `adder`'s result in different cycles when $\neg con$. Thus, a refinement mapping does not exist.

When $s = Super$, `adder`'s output wires and result buffer are now asserted and written to by `half2` and `or`. These are produced in different cycles, thus the refinement mapping does not exist.

A second attempt: hide everything

The verification failure follows from a lack of constraints in the TLA embedding of the Hume program. The most constrained embedding involves hiding all variables, thus $Trans(P_1, P_2)$ becomes

$$(\exists\, ws, inp, res, st, s, pc, con, expr.\ I_2 \wedge \Box[\mathcal{N}_2]_w) \\ \Rightarrow (\exists\, ws, inp, res, st, s, pc, con, expr.\ I_1 \wedge \Box[\mathcal{N}_1]_v).$$

The transformation is verified by "slowing down" the input consumption of the spatial component until the cycle where `half2` executes, and output is computed and written to when `or` executes. This is achieved by introducing *history variables* to P_2 which

buffer and reset the "actual" values, and the refinement mapping $\overline{\cdots}$ is defined over these. In the consume step, $\overline{iws_{\texttt{adder}}}$ and $\overline{inp_{\texttt{adder}}}$ are updated when `half2` successfully executes. This mapping refers to both `half1` and `half2` (via the history variable). In the compute step, $\overline{res_{\texttt{adder}}}$ is updated when `or` executes. $\overline{ows_{\texttt{adder}}}$ is updated in the super-step where `or` actually asserts and writes to the output.

In addition to these local properties for the resulting spatial component, the refinement mapping must reflect global properties and an "initialisation phase" for the first few cycles. This phase allows the first values to go through all boxes, before values are produced in every other cycle by the `show` box. Globally, in P_1, after the initialisation phase, all three boxes match the input and produce output in all cycles, such that at cycle n `gen` produces output to `adder`; these values are input to `adder` at cycle $n+1$, where the results are outputted to `show` in the same cycle; which produces the result in cycle $n+2$. In P_2 on the other hand, these values are not consumed by `show` before the $(n+4)^{th}$ cycle. Moreover, `gen` is *Blocked* in every other cycle, while `show` will *Matchfail* in every other cycle. These global properties are also captured by the history variables and refinement mapping, and the example has been verified by TLC under this refinement mapping.

Discussion

The example illustrates the difficulties with verifying Hume transformations. It shows that (complex) global analysis is required for the verification, and this is particular hard in Hume, since the coordination layer is flat, and thus does not contain any support for structuring. Consequently, all boxes in a program must be analysed.

Part of the problem in the example was that input and output were consumed and produced by different boxes in different cycles. In general, having distinct "first" and "last" boxes for a spatial component, as illustrated in Figure 5.6 where `D1` is the "first" box and `D4` is the "last box", does not solve the problem. For example, if box `E` has the body

```
box E ...
 ... | (x,y) -> g x y | (*,y) -> f y ;
```

then the source program may trigger the first match producing the `g x y` result. The extra cycles of the resulting spatial `D1` to `D4` component, compared to input from box `C`, may cause the first match to fail, and trigger the second match, thus `E` produces the `f y` result. Thus, change of coordination behaviour may change the functional behaviour. Thus, although *functional properties* are preserved by the transformed component, the *coordination properties* may have changed the whole system radically.

5.6 Summary & discussion

This chapter has discussed the verification of invariants and transformation of Hume programs. Tactics have been developed to automate invariant proofs in Isabelle/Hume, and verification by both model checking and theorem proving has been illustrated by a set of case-studies. The model checking case study in Section 5.5.1 also illustrates verification of action/computation and real-time properties.

TLA$^+$, and thus TLA, have been previously used in larger examples [73] and industry [24]. However, the work here is novel by in that it applies TLA to a programming language. In the B-method, which is quite similar to TLA, but without the temporal level, specifications are refined to such a low level, that at the end, a C or Ada program can be synthesised. Similarly, in [181], temporal logic is applied to synthesise embedded programs. In TLA, the only known similar work, is some minor unpublished notes on translating TLA into Ada programs [113]. The property verification by model checking, as discussed in case study H1 (Section 5.5) and published in [91], is on the HW-Hume level, which attempts to model low-level hardware systems in the high-level Hume notation. Similar properties for hardware systems have been verified using dependent types [30].

The structure of Hume programs (boxes and wires), and shallow embedding, have been explored by developing tactics to automate the verification, and by heavily using built-in Isabelle tools. One example (the vending machine) illustrated the problem with this rather uncontrolled rewriting, since it failed on a proof. This is discussed further in Chapter 7. Moreover, the example illustrated the problem of specifying properties over a cluster of boxes, where many interesting properties lie. This requires the introduction of auxiliary variables, used to specify partial correctness properties. This is used in, for example, Hoare logic. However, the sequential imperative programs, do not require the "sequentiality proof", where in the vending example, the tactics failed. Moreover, such properties will not be preserved by a transformation, since the auxiliary variables are not part of the specification.

The work with program transformation using TLA is believed to be novel. The verification approach taken, bears similarities to the *reduction theorem* [120, page 41], used to prove when a finer-grained program implements a coarser grained one. This can be used to turn a critical section into an atomic step. Here, all variables are $\exists$ bound and a predicate $\Box R$, which must be verified, links the "real variables" and refinement mapping. This seems similar to the linking invariant used in the B method, albeit establishing how similar, requires more work. Note, that refinement, and thus Hume transformations, can be verified by model checking in B with the ProB [133] tool. In a process algebra, the transformation can be verified as a bisimulation.

In this chapter the problems with transformation and reasoning over many boxes have been illustrated. This is mainly down to a lack of structuring in the coordination layer. In the next chapter, an extension to Hume, *Hierarchical Hume*, is introduced which solves these problems.

Chapter 6

Hierarchical Hume[1]

"Nothing is particularly hard if you divide it into small jobs."

– Henry Ford

6.1 Introduction

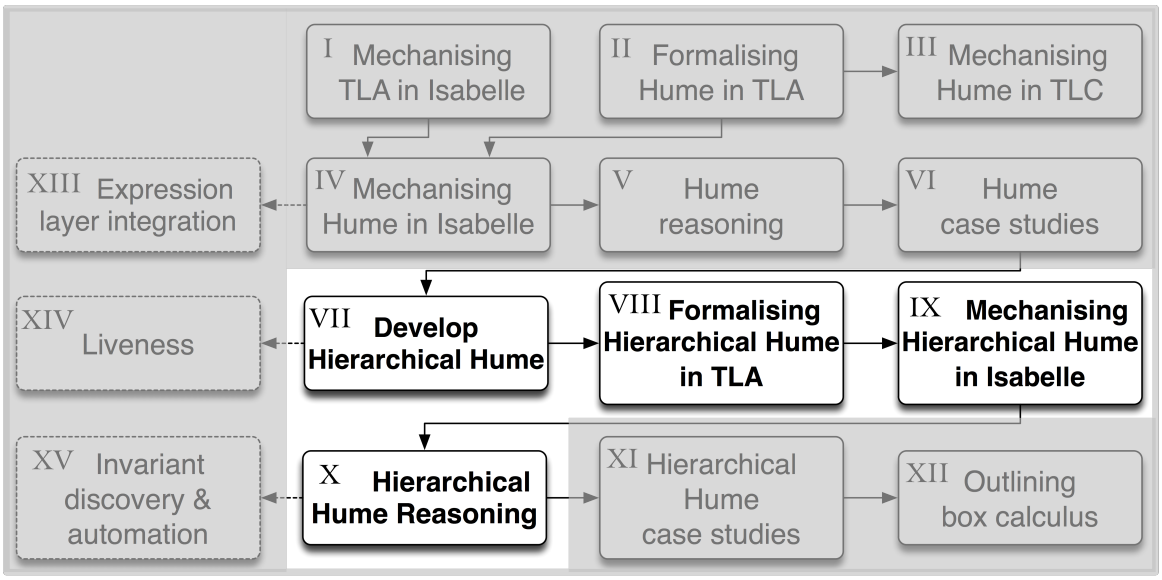


Figure 6.1: Thesis roadmap: Chapter 6

The previous two chapters described formal verification of Hume programs using TLA, and revealed difficulties for both the specification and verification of coordination layer properties in the current form. This chapter proposes an extension to Hume, called

[1]Section 6.1.2 and Section 6.2 has been developed jointly with Robert Pointon. He has extended the Hume interpreter with a prototype Hierarchical Hume implementation. This has enabled testing of programs, and provided empirical evidence showing that this extension works.

Hierarchical Hume, which solves these difficulties. This is achieved by allowing boxes to be nested within other boxes, thus introducing box hierarchies. The semantics of Hierarchical Hume is both formalised in TLA and mechanised in Isabelle/TLA. Moreover, automation of invariant and transformation verification is discussed and implemented. Figure 7.1 highlights which parts of the roadmap this chapter implements. Firstly, the motivations and other outcomes are summarised.

6.1.1 Motivations

Most *program transformations* are a result of a failed costing, which is always in the expression layer. The result is to move computation into the coordination layer. Since the expression layer can only communicate over the coordination layer, such transformations always involve one box being transformed into one or more boxes and wires. The previous chapter showed that the changes imposed on the coordination layer by these transformations could change the overall program behaviour and require full global analysis. The nesting of boxes enables such transformations without changing the structure of the coordination layer.

Moreover, non-trivial coordination properties are often over a cluster of boxes and wires, which together produce a computation, or a box that produces the result over may cycles. For example, in the vending-machine example (Section 5.5.3), the result of three boxes resulted in a computation, while the `mult` box (Section 4.2) produced the result over several cycles. *Asserting* these properties requires feedback wires to stop new computations starting and auxiliary TLA variables. From a logical viewpoint auxiliary variables can be used for verification, but not specification since they are not part of the model. Hierarchical Hume provides a natural way of specifying assertions of a (nested) cluster of boxes.

Now, Hierarchical Hume enables *local reasoning* of both transformations and invariants. This becomes important in both model checking and theorem proving large programs, since the search space can be reduced. Moreover, it enables better integration of these techniques, where a theorem proven is used for the overall proof, while (nested) box lemmas can be solved by model checking. Liveness properties, partly discussed in Chapter 10, may depend on the blocking status of the relevant box, which may depend on the blocking status of other boxes and so on. Thus, in these cases local reasoning becomes very important.

6.1.2 Further motivations and positive outcomes

Hume supports *structuring* via the layered architecture, where the expression layer is independent of all other boxes. However, only two layers (coordination and expression) are allowed. Thus, a large program will consist of one very large low-level coordination layer of (dependent) boxes, which is hard to work with and error-prone. Hierarchical Hume allows logically connected boxes to be composed, thus supporting many layers, and a better program structure.

The issue with unnecessary box matching, which was solved by self-out *scheduling* in Section 4.2, can also be handled by the user by nesting boxes with coordination iteration. This is a more general solution than self-out scheduling which gives the user more control.

Hume offers standard *library* functions for the expression layer. In particular, the *Template-Hume* level contains a set of pre-costed higher-order functions like `map` and `foldl`. Hierarchical Hume enables a library of pre-costed (possibly nested) boxes. Such a library will support a *software product line* [44], where pre-costed components can be plugged together to form a program. This fits into the *costing-by-construction* principle [175] introduced for Hume development.

By executing various parts of a program in *parallel* [60], the performance is often increased, in particular now since most processors are multi-core. In Hume single boxes can easily be parallelised. However, due to box dependency and strict scheduling, a cluster of boxes cannot in general be independently scheduled, meaning parallelism of clusters becomes hard. The nesting of boxes enables independent scheduling, thus enabling direct parallelism of a cluster of boxes.

6.2 Overview of Hierarchical Hume

Let *flat* Hume be the Hume version used so far. A *nesting* box is a box that contains other boxes, and it is the *parent* of the nested boxes, which are its *children*. Boxes with the same parent are *siblings*, and a box can only be wired to its siblings, parent and children. A non-nesting box is *flat*, and is identical to a box in flat Hume. A nesting box is mainly a scheduling abstraction, which allows a subset of (child) boxes to be scheduled independently. When ignoring types and children, a nesting box is in the HW-Hume level, with some minor (non-computational) extensions. Hierarchical Hume should support existing Hume tools, thus a nesting box is still an input-output relation, without any dangling state between calls, and no side-effects. However, initial wire values which are reset between each call, are allowed.

A nesting box is executed by passing the "scheduling token" to it. If the input

pattern matches, the children are scheduled until termination, and the token returned. Since the children may be nested, this scheduling is recursive. If the pattern matching succeeds, the match's input wires are copied to "internal input wires" and when it terminates "internal output wires" are returned. These wires are those which are wired between the parent and children. The programmer must define the start and termination of a nesting box, and the match syntax is exploited, with the meaning '*pattern* `->` *termination condition*', where the *pattern* also defines which values that are copied to the "internal input wires". The *termination condition* refers to "internal output wires", and also defines which values are returned.

box	::=	`box` id *ins outs* `match` $\langle$ $bmatch_1$ $\mid$ $\cdots$ $\mid$ $bmatch_n$ $\rangle$	$n \geq 1$
	$\mid$	`box` id *ins outs* `match` *hmatch* `boxes` *program* `end`	
hmatch	::=	$\langle$ $bpat_1$, $\ldots$, $bpat_n$ $\rangle$ `->` *hexpr*	$n \geq 1$
hexpr	::=	int $\mid$ bool $\mid$ word $\mid$ con $\mid$ `*` $\mid$ `_` $\mid$ `(` $\langle$ $hexpr_1$, $\ldots$, $hexpr_n$ $\rangle$ `)`	$n \geq 1$

Figure 6.2: Hierarchical Hume abstract syntax extension

Figure 6.2 extends the TLA-Hume abstract syntax from Figure 2.8 (page 35) with nesting boxes. Now, a *pattern* for a nested box is the same as a pattern for a flat box. An *expression* in a nested box is either a constant, '`*`', '`_`', or a tuple of expressions. Here, as in a pattern, '`_`' denotes the existence of a value. In the body (*program*) of a nesting box, boxes, wires and functions can be declared. Here, all input wires to the parent box, must be wired to one or more of the children, while all output wires of a nesting box must be wired from one or more of the children.

A nesting box behaves like a scheduler for its children. This *hierarchical scheduling* is achieved by nesting the lock-step scheduling from Section 4.2, with an additional check for termination. This requires changes in the possible states a box may be in after each phase, resulting in six possible states:

- *Runnable*: the box has asserted output and is ready to consume input;
- *Blocked*: the box has successfully consumed inputs but failed to assert outputs. It will attempt to assert outputs on subsequent cycles;
- *Matchfail*: the box has failed to match inputs;
- *Terminated*: the box has finished executing and is ready to super-step;
- *Execute*: the box is ready to execute its children;

- *Super*: the box is ready to super-step its children.

The *Blocked* and *Matchfail* states are unchanged from Section 4.2, while the remaining steps deviate from Section 4.2 as follows: the lock-step *Runnable* state is split into *Runnable* and *Terminated*; while the new *Execute* and *Super* states are used to schedule sub-components. The hierarchical scheduling is then recursively given as follows:

```
schedule (boxes, condition, result) ≜
  until condition(boxes)
    for each Runnable or Matchfail box in boxes
      if box is nesting
        then schedule(children(box), termination_condition(box))
        else execute(box)
    super_step (each Terminated or Blocked box in boxes)
  copy(output_wires(boxes), result)
```

This *schedule* operation receives a set *boxes* and a predicate *condition* on the *boxes* set. This predicate represents the termination condition. The assumed *termination_condition* function on a box returns this predicate. As long as *condition(boxes)* evaluates to *False*, all the boxes are scheduled. When it evaluates to *True*, the internal output wires, returned by the *output_wires* function over a box set, are then copied to the given *result* buffer. The first-level scheduler never terminates, and is thus defined as

$$schedule\big(\text{all top-level boxes}, (\lambda b.\ False)\big)$$

Above, the *children* function returns the set of children boxes for a given nesting box, while *execute* behaves the same as in Section 4.2.

Figure 6.3 illustrates the source code and box diagram of a nesting box `mult`. It accepts two inputs, which are multiplied by iteration using the nested `itermult` box. The result is produced in one box `mult` step, and the boxes are in the FSM-Hume level. The wiring between the `mult` (parent) and the `itermult` (child) boxes is defined in the wire declaration of `itermult`.

6.3 Hierarchical Hume formalised in TLA

The body of a nesting box, which is the internal wires and boxes, can either be part of the specification, or internal to the specification. The first case represents a "black-box" embedding, whilst the latter enables reasoning about "nested properties", that is, properties of nested boxes and wires. Logically, the first approach is separated from

```
type Nat = word 64;

box mult
 in (a,b::Nat)
 out(o::Nat)
match
  (_,_) -> (_)
boxes

  box itermult
    in (a,b,iter::(Nat,Nat,Nat))
    out (o,iter'::(Nat,Nat,Nat))
    match
      (a,b,   *   )->(*,( 0 ,a, b ))
    | (*,*,(r,x,0))->(r,     *     )
    | (*,*,(r,x,y))->(*,(r+x,x,y-1));

  wire itermult (mult.a,mult.b,itermult.iter') (mult.o,itermult.iter);
end;
```

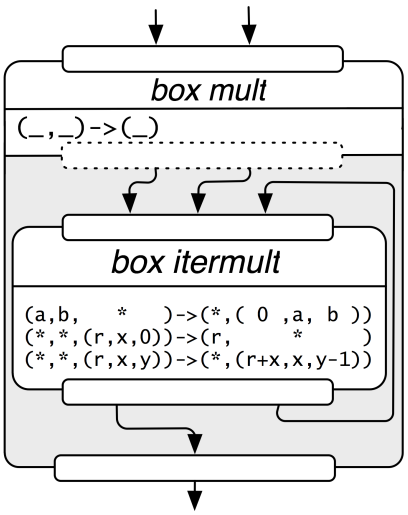


Figure 6.3: Hierarchical Hume example

the latter by ∃-binding the internal wires and box components. Since reasoning about nesting boxes is a desired feature, the box body is seen as part of the specification, hence the nested boxes and wires are free. Now, let the total function $children \in BS \to 2^{BS}$ return a set containing the immediate children identifiers of the given box. That does not include potential nested children. Note that BS contains all the box identifiers, including nested boxes. From the *children* function, the following functions are derived:

$$\begin{array}{llll} first\_level(i) & \triangleq & \forall j \in BS.\ i \notin children(j) & flat(i) \triangleq children(i) = \{\} \\ BS_1 & \triangleq & \{b \in BS.\ first\_level(b)\} & nested(i) \triangleq \neg flat(i). \end{array}$$

first_level is a predicate holding if the box is a first-level (non-nested) box, and $BS_1 \subseteq BS$ is the set of all first-level boxes. A box is *flat* if it does not have any children, and *nested* if it has children. The syntax ensure that a nesting box must have at least one child. Let s remain the first level scheduler. From this the following functions are derived:

$$\begin{array}{lll} parent(i) & \triangleq & \epsilon\ j \in BS.\ i \in children(j) \\ siblings(i) & \triangleq & \textbf{if } first\_level(i) \textbf{ then } BS_1 \textbf{ else } children\big(parent(i)\big) \\ boxsch(i) & \triangleq & \textbf{if } first\_level(i) \textbf{ then } s \textbf{ else } st_{parent(i)} \end{array}$$

parent is a partial function which returns the parent of the given box; *siblings* returns the set of boxes with the same parent; while *boxsch* returns the scheduling value of a given box, which is either the state variable of the parent or s. The (immediate) internal wires Iws_i of a nesting box are divided into three disjointed tuples: $Iiws_i$ is

the internal input wires; $Iows_i$ is the internal output wires; and $IIws_i$ is the remaining internal wires. $Iiws_i$ and $Iows_i$ are assumed to have the same order as the corresponding input buffer inp_i and result buffer res_i. Iws_i is defined as $Iws_i \triangleq Iiws_i \cdot Iows_i \cdot IIws_i$, while $flat(i) \Rightarrow Iws_i = \langle\rangle$.

A nested box terminates when the termination condition is met, that is when the following predicate holds for the box expression e and internal output wires ows:

$$
\begin{array}{l}
wire\_terminated(ow, e) \triangleq \\
\quad (e = *) \vee (e = \_ \wedge ow \neq \bot) \vee (e \in \{\text{int,bool,word,con}\} \wedge e = ow) \\
box\_terminated(ows, e) \triangleq \\
\quad ows = \langle\rangle \vee \Big(wire\_terminated\big(hd(ows), hd(e)\big) \wedge box\_terminated\big(tl(ows), tl(e)\big)\Big)
\end{array}
$$

Note that $*$ is used in the expression of the nesting box to describe a wire in which *wire_terminated* will always succeed, while $\bot$ is an empty value still. A box is terminated T, when the state st has the value:

$$T(st) \triangleq st \in \{Terminated, Matchfail, Blocked\}$$

To enable transformation proofs, the more concrete two step sequential box execution described in Section 5.4, is formalised. Moreover, due to the nesting of boxes, the scheduler is also nested. This requires a redefinition of *first_box*, *last_box* and *next_box* to a given set of boxes $bs \subseteq BS$, instead of all the boxes BS:

$$
\begin{array}{lcl}
first\_box(bs) & \triangleq & \epsilon\, pc \in bs.\ \forall p \in bs.\ pc \neq p \Rightarrow pc \prec p \\
last\_box(bs) & \triangleq & \epsilon\, pc \in bs.\ \forall p \in bs.\ pc \neq p \Rightarrow p \prec pc \\
next\_box(pc,bs) & \triangleq & \textbf{if}\ \ pc = last\_box(bs) \\
 & & \quad \textbf{then}\ pc \\
 & & \quad \textbf{else}\ \epsilon\, p \in bs - \{pc\}.\ \forall q \in bs - \{pc, p\}.\ pc \prec p\ \wedge\ p \prec q
\end{array}
$$

Since the scheduler is nested it is not only defined for s, but accepts the current scheduler as input. Hence, it is a function S and not an action $\mathcal{S}$, and is defined as:

$$
\begin{array}{l}
S(sch, pc, terminated, bs) \triangleq \\
\quad \textbf{case}\ \ sch = Execute \wedge \big(pc = last\_box(bs) \wedge terminated\big) \rightarrow Super \\
\quad \Box\ \ sch = Super \wedge terminated \rightarrow Terminated \\
\quad \Box\ \ \textbf{else} \rightarrow Execute.
\end{array}
$$

It accepts: the (old) scheduling value *sch*; the (current) program counter *pc*; a boolean *terminated* which hold if the (parent/scheduler) box is terminated in the *Super* step

and the current (pc) box is terminated in the *Execute* step. The extra possible box states absorb the need for the *con* variable used in Section 5.4. Finally, bs is the set of boxes (children) that are scheduled. In the *Execute* phase the scheduler changes to *Super* when the last box has terminated, while in the *Super* phase it changes to *Terminated* if the box (scheduler/parent) has terminated. In all other cases *Execute* is returned: $hexecute_f^{con}$ and $hexecute_n^{con}$ update $execute^{con}$ for a flat and nested box, respectively:

$$\begin{aligned}
&hexecute_f^{con}(rs,iws,e) \triangleq \\
&\quad \epsilon\ \langle w,inp,ne,st\rangle. \quad \big(\exists\, ost.\ \langle w,inp,ne,\, ost\rangle = execute^{con}\big(rs,\, iws,\, exp(hd(rs))\big) \\
&\qquad\qquad \wedge\ st = (\textbf{if}\ ost = Runnable\ \textbf{then}\ Execute\ \textbf{else}\ ost)\ \big) \\
&hexecute_n^{con}(rs,iws) \triangleq \\
&\quad \epsilon\ \langle w,inp,inp,st\rangle. \quad \big(\exists\, e_1, e_2.\ \langle w,inp,st,e_2\rangle = hexecute_f^{con}(rs,iws,e_1)\big)
\end{aligned}$$

$hexecute_f^{con}$ replaces the *Runnable* state with *Execute*, required to avoid using the *con* variable; $hexecute_n^{con}$ ignores the expression since a nested box can only have one. Moreover, two copies of the input values are returned: one for the input buffer inp_i; and one for the internal input wires $Iiws_i$ for $i \in BS$ and $nested(i)$.

Based on these functions, all the actions formalising Hierarchical Hume can be defined. Firstly, the action $\mathcal{U}_i$ leaves all the box variables, included nested (at all depths) components, unchanged:

$$\begin{aligned}
\mathcal{U}_i \triangleq\ & (boxsch(i) = Execute \Rightarrow iws_i' = iws_i)\ \wedge\ (boxsch(i) = Super \Rightarrow ows_i' = ows_i) \\
\wedge\ & \langle st_i, inp_i, res_i\rangle' = \langle st_i, inp_i, res_i\rangle \\
\wedge\ & \big(nested(i) \Rightarrow (\textstyle\bigwedge_{j \in children(i)} \mathcal{U}_j) \wedge \langle pc_i, Iws_i\rangle' = \langle pc_i, Iws_i\rangle\big)
\end{aligned}$$

$h\mathcal{B}_i^s$ extends the super-step box action $\mathcal{B}_i^{s^2}$ with explicitly unchanging nested components: $h\mathcal{B}_i^s$:

$$h\mathcal{B}_i^s \triangleq\ \mathcal{B}_i^{s^2} \wedge \big(nested(i) \Rightarrow \langle pc_i, Iws_i\rangle' = \langle pc_i, Iws_i\rangle \wedge \textstyle\bigwedge_{j \in children(i)} \mathcal{U}_j\big).$$

Due to the nesting, each nesting box needs its own program counter. Thus, pc is now a tuple holding all the program counters in an order based on $\prec$ of the box identifiers. However, pc_1 is assumed to be the first-level scheduler, that is pc in the formalisation of flat Hume. $h\mathcal{B}_i^{ef}$ is the box execute action for a flat box. The extra potential box states remove the need for *con* to separate the consume and compute step. Moreover, with respect to $\mathcal{B}_i^{e^2}$, the call to $execute^{con}$ is now replaced with $execute_f^{con}$, and the "guards"

are over all the possible state variables:

$$
\begin{array}{lll}
h\mathcal{B}_i^{ef} \triangleq & st_i = \mathit{Runnable} & \Rightarrow \langle iws_i, inp_i, expr, st_i\rangle' = execute_f^{con}(rs_i, iws_i, expr) \\
& & \wedge \quad res_i' = res_i \\
\wedge & st_i = \mathit{Execute} & \Rightarrow \langle expr, iws_i, inp_i, res_i\rangle' = \langle expr, iws_i, inp_i, hrun(expr, inp_i)\rangle \\
& & \wedge \quad st_i' = \mathit{Terminated} \\
\wedge & T(st_i) & \Rightarrow \langle expr, iws_i, inp_i, res_i, st_i\rangle' = \langle expr, iws_i, inp_i, res_i, st_i\rangle.
\end{array}
$$

Note that st_i cannot be *Super* for a flat box $i \in BS$, i.e. $\Box\big(\mathit{flat}(i) \Rightarrow st_i \neq \mathit{Super}\big)$. A proof of this is given in Appendix B.2.

$$
\begin{array}{lll}
h\mathcal{B}_i^{en} \triangleq & st_i = \mathit{Runnable} & \Rightarrow \langle iws_i, inp_i, Iiws, st_i\rangle' = hexecute_n^{con}(rs_i, iws_i) \\
& & \wedge\ res_i' = res_i \wedge expr' = expr \\
& & \wedge\ \langle IIws_i, Iows_i\rangle' = \langle I_{IIws_i}, I_{Iows_i}\rangle \\
& & \wedge\ pc_i' = \mathit{first\_box}\big(\mathit{children}(i)\big) \ \wedge\ \bigwedge_{j\in \mathit{children}(i)} \mathcal{R}_j \\
\wedge & st_i = \mathit{Execute} & \Rightarrow h\mathcal{B}^e_{pc_i} \wedge \bigwedge_{j\in \mathit{children}(i)-pc_i} \mathcal{U}_j \\
& & \wedge\ pc_i' = \textbf{if } T(st'_{pc_i}) \textbf{ then } \mathit{next\_box}\big(pc_i, \mathit{children}(i)\big) \textbf{ else } pc_i \\
& & \wedge\ st_i' = S\big(st_i, pc_i', T(st'_{pc_i}), \mathit{children}(i))\big) \\
& & \wedge\ \langle iws_i, inp_i, res_i, Iows_i\rangle' = \langle iws_i, inp_i, res_i, Iows_i\rangle \\
\wedge & st_i = \mathit{Super} & \Rightarrow \big(\bigwedge_{j\in \mathit{children}(i)} h\mathcal{B}^s_j\big) \wedge\ pc_i' = \mathit{first\_box}\big(\mathit{children}(i)\big) \\
& & \wedge\ st_i' = S\big(st_i, pc_i', \mathit{box\_terminated}(Iows_i', \mathit{exp}(\mathit{hd}(rs_i))), \\
& & \qquad\qquad \mathit{children}(i)\big) \wedge expr' = expr \\
& & \wedge\ res_i' = \textbf{if } st_i' = \mathit{Terminated} \textbf{ then } Iows_i' \textbf{ else } res_i \\
& & \wedge\ \langle iws_i, inp_i, Iiws_i\rangle' = \langle iws_i, inp_i, Iiws_i\rangle \\
\wedge & T(st_i) & \Rightarrow \mathcal{U}_i \wedge expr' = expr
\end{array}
$$

Figure 6.4: $h\mathcal{B}_i^{en}$: TLA action for execute phase of nested Hume box

The box action $h\mathcal{B}_i^{en}$ for the execute phase of a nested box is shown in Figure 6.4. It is defined by case-analysis on the box state st_i.

In the *Runnable* state, the box attempts to match and consume inputs using $hexecute_n^{con}$. Moreover, the internal non-input wires are set to the initial values, given by I_{IIws_i} and I_{Iows_i}, and the program counter is set to the first box. All the children are reset to start executing, achieved by the $\mathcal{R}_i$ action:

$$
\begin{array}{ll}
\mathcal{R}_i \triangleq & \langle st_i, inp_i, res_i\rangle' = \big\langle \mathit{Runnable}, \langle \bot, \cdots, \bot\rangle, \langle \bot, \cdots, \bot\rangle\big\rangle \\
\wedge & (\mathit{nested}(i) \Rightarrow Iws_i' = Iws_i \ \wedge\ \bigwedge_{j\in \mathit{children}(i)} \mathcal{U}_j);
\end{array}
$$

In the *Execute* state, the current box pc_i , is executed using $h\mathcal{B}^e_{pc_i}$, which abstract

over whether or not pc_i is nested:

$$h\mathcal{B}_i^e \triangleq \ (\mathit{nested}(i) \Rightarrow h\mathcal{B}_i^{e_n}) \wedge (\mathit{flat}(i) \Rightarrow h\mathcal{B}_i^{e_f}).$$

Due to the two-step execution and the fact that the box's children may be nested, pc_i is only updated when the current box terminates, i.e. $T(st'_{pc_i})$ holds. Moreover, S is used to obtain the new value of st_i, with the obvious inputs.

In the *Super* state, all children boxes are executed, and pc_i is reset back to the first box. S is used to check for termination, and if box i has terminated, the result buffer is updated. Note that in the *Super* and *Execute* states, box i behaves as the environment in flat Hume: the input and output wires are unchanged, since none of the child boxes "own"" them. Finally, if the box is terminated, or either *Blocked* or *Matchfail*, all variables "belonging to" box i are unchanged.

The first-level next state action is a simpler version of $h\mathcal{B}_i^{e_n}$:

$$\begin{array}{rlll} h\mathcal{N} \triangleq & \big(s = \mathit{Super} & \Rightarrow & (\bigwedge_{i \in BS_1} h\mathcal{B}_i^s) \wedge pc'_1 = \mathit{first\_box}(BS_1 \cup \{\mathit{env}\}) \\ & & \wedge & s' = S(s, pc_1, \mathit{False}, BS_1 \cup \{\mathit{env}\}) \wedge \mathit{expr}' = \mathit{expr}\big) \\ \wedge & \big(s = \mathit{Execute} & \Rightarrow & (pc_1 \in BS_1 \Rightarrow h\mathcal{B}^e_{pc_1}) \wedge \bigwedge_{i \in (BS_1 - \{pc_1\})} \mathcal{U}_i \\ & & \wedge & pc'_1 = \textbf{if } pc_1 = \mathit{env} \vee T(st'_{pc_1}) \\ & & & \qquad \textbf{then } \mathit{next\_box}(pc_1, BS_1 \cup \{\mathit{env}\}) \textbf{ else } pc_1 \\ & & \wedge & s' = S(s, pc'_1, pc'_1 = \mathit{env} \vee T(st'_{pc}), BS_1 \cup \{\mathit{env}\})\big). \end{array}$$

Initially, an Hierarchical Hume program has the following state:

$$\begin{array}{rl} hI \triangleq & ws = w_I \wedge s = \mathit{Execute} \wedge pc_1 = \mathit{first\_box}(BS_1 \cup \{\mathit{env}\}) \\ \wedge & \bigwedge_{i \in BS} \big(st_i = \mathit{Runnable} \wedge \langle \mathit{inp}_i, \mathit{res}_i \rangle = \big\langle \langle \bot, \cdots, \bot \rangle, \langle \bot, \cdots, \bot \rangle \big\rangle\big) \\ \wedge & \bigwedge_{i \in BS} \big(\mathit{nested}(i) \Rightarrow pc_i = \mathit{first\_box}(\mathit{children}(i))\big) \wedge \mathit{expr} = \langle \bot, \cdots, \bot \rangle \end{array}$$

Note that ws includes the nested wires. $h\mathcal{N}$ subsumes both the next box action and first-level scheduler. Finally, $h\mathcal{E}$ updates $\mathcal{E}$ by using pc_1 instead of pc:

$$\begin{array}{rll} h\mathcal{E} \triangleq & (s = \mathit{Execute} \wedge pc_1 = \mathit{env} & \Rightarrow \forall w \in \mathit{oout\_ws} : w' = w \vee w' = \bot) \\ \wedge & (s = \mathit{Execute} \wedge pc_1 \neq \mathit{env} & \Rightarrow \forall w \in \mathit{ows} : w' = w) \\ \wedge & (s = \mathit{Super} & \Rightarrow \forall w \in \mathit{in\_ws} : \exists v \in T \cup \{\bot\} : w' = v) \end{array}$$

The full program H_{hi} is thus defined as follows:

$$\begin{array}{rcl} H'_{hi} & \triangleq & hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E}]_{\langle s, ws, inp, res, st, pc, expr \rangle} \\ H_{hi} & \triangleq & \exists\, st, s, pc, expr.\ H'_{hi} \end{array} \tag{6.1}$$

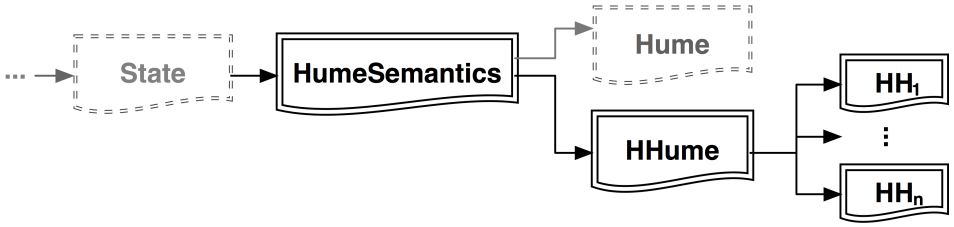


Figure 6.5: Theory structure of Hierarchical Hume embedding Isabelle/TLA

From the syntax definition in Figure 6.2 it is clear that syntactically, the Hierarchical Hume extension is conservative. However, it is also important to show that this is also the case semantically. Thus, it must be verified that the behaviour of the two-phase box execution of flat Hume from Section 5.4 is preserved. However, flat Hume does not support box nesting, reducing the property to '*if no boxes are nested, then hierarchical scheduling preserves the behaviour of lock-step scheduling*'. The property is formalised in Theorem 6.1, and requires Lemma 6.1.

Lemma 6.1. *If* $\bigwedge_{i \in BS} flat(i)$ *and* $BS_1 = BS$ *then*

1. $first\_box(BS_1 \cup \{env\}) = first\_box$;
2. $last\_box(BS_1 \cup \{env\}) = last\_box$;
3. $next\_box(pc, BS_1 \cup \{env\}) = next\_box(pc)$;
4. $h\mathcal{B}_i^e \equiv h\mathcal{B}_i^{ef}$.

Proof outline. The proof follows directly from unfolding the assumptions. ∴

Theorem 6.1. *If* $\bigwedge_{i \in BS} flat(i)$ *then* H_{hi} *implies* H_{seq^2}.

Proof outline. The proof relies on the invariants $\Box(s \in \{Execute, Super\})$, $\Box(\forall i \in BS.\ st_i \neq Super)$, $\Box(s = Execute \Rightarrow \forall p \in BS_1.\ pc_1 \preceq p \Rightarrow st_p \in \{Runnable, Blocked, Execute\})$ and $\Box(s = Execute \Rightarrow \forall p \in BS_1.\ pc_1 \prec p \Rightarrow st_p \in \{Runnable, Blocked\})$. It uses the refinement mappings $\overline{st_i} \triangleq$ **if** $st_i \in \{Terminated, Execute\}$ **then** $Runnable$ **else** st_i and $\overline{con} \triangleq pc_1 = env \vee st_{pc_1} \neq Execute$. The proof uses the (E2), (E1), (TLA2) and (STL4) and follows mainly by case-analysis. ∴

The full proofs of the theorem and required invariants are given in Appendix B.2.

6.4 Hierarchical Hume in Isabelle/TLA

The mechanisation of Hierarchical Hume (Isabelle/HHume) in Isabelle/TLA is based on the Hume mechanisation from Section 4.5. It is shallow, and pc is updated inside

the boxes. This enables use of both one- and two-step box execution for flat boxes. To enable more direct reuse of Section 4.5, all boxes that are not transformed are given a one-step box execution. The theory structure is shown in Figure 6.5: HHume mechanises commonalities of Hierarchical Hume, and re-uses the HumeSemantics theory. All embedded Hierarchical Hume programs use the HHume theory.

6.4.1 The HHume theory

Since the possible box states are not separated from the possible scheduling states in Hierarchical Hume, both these states are represented by the same enumeration type

```
datatype box_state = Runnable | Blocked | Matchfail
                   | Execute | Super | Terminated .
types boxstate = box_state statefun
```

S mechanises the scheduling function S as formalised in Section 6.3:

```
S                          :: box_state ⇒ bool ⇒ bool ⇒ bool ⇒ BoxState
S Execute True True  _     = Super
S Super     _     _  True  = Terminated
S  _        _     _   _    = Execute .
```

Due to the shallow embedding, *pc* depends on the program, and the box identifier sets are not applied directly. Instead, the second argument of S (*pc* in S) is a predicate that must hold if *pc* is the last box. Further, in S, *terminated* has two different meanings depending on the current value of the scheduler. This is split here, thus introducing an additional argument. This enables a first-level scheduler S1:

```
S1           :: box_state ⇒ bool ⇒ bool ⇒ box_state
S1 sch pc t  = S sch pc t False .
```

The following scheduling lemmas are required in the reasoning:

```
lemma sch1:   S b pc t psch ∈ {Super,Execute,Terminated}
lemma sch2:   S Execute pc t psch ∈ {Super,Execute}
lemma sch3:   S Super pc t psch ∈ {Terminated,Execute}
lemma sch4:   (S b pc t psch = Super) ∨ (S b pc t psch = Terminated) .
lemma S1sch1:  S1 s pc t ∈ {Super,Execute}
```

Proof outline. sch1 by induction on b followed by case-analysis. The other lemmas by case-analysis. ∴

The first-level scheduler s is of type boxstate:

```
s  :: boxstate .
```

Finally, the termination condition function is also defined by pattern matching:

```
termcond                :: box_state ⇒ bool
termcond Terminated     = True
termcond  Matchfail     = True
termcond   Blocked      = True
termcond    _           = False ,
```

and it can also be represented as a set:

lemma `termcond1:` termcond x = (x ∈ {Terminated,Blocked,Matchfail})

Proof outline. By induction on x. ∴

6.4.2 An example embedding

The mechanisation is illustrated by the program shown in Figure 6.3. It is assumed that the input and output wires of `mult` are connected to streams, while `itermult` is represented as a flat box, which is already explained in Section 4.5 and not given here. Firstly, the PC type and pc variable are defined:

```
datatype PC = pinc | penv
pc :: PC statefun .
```

S' specialises the first-level scheduler for this particular program:

```
S' ≡   s$ = S1<$s, $pc ≠ #penv,#True>.
```

Note that the last argument is True since the environment is the last box which terminates in one step. The program contains three first level wires, and one first level box `mult`:

```
w1, w2, w3   :: nat HVal
mult_inp     :: ((nat option)×(nat option)) statefun
mult_res     :: nat HVal
mult_st      :: boxstate.
```

Internally the `mult` box contains four wires:

```
iw1, iw2, iw4   :: nat HVal
iw3             :: (nat × nat × nat) HVal,
```

where iw1 and iw2 are the internal input wires, iw3 the feedback loop of the `itermult` box, and iw4 the internal output wire. Moreover, a program counter type EPC and

nested program counter epc ($pc_{\texttt{mult}}$) are defined. Since there is only one nested box this is strictly not required, however it is required in general when more than one box is nested, and is thus included here:

```
datatype EPC = epitermult
epc :: EPC statefun .
```

Let itermult be the box action for the `itermult` box, consisting of the itermult_exe and itermult_sup actions. Since `itermult` is nested by `mult`, the state of `mult` behaves as the scheduler for `itermult`:

```
itermult  ≡   ($mult_st = #Execute  ⟶  itermult_exe)
          ∧   ($mult_st = #Super    ⟶  itermult_sup).
```

itermult, and it's actions uses the internal wires and `epc`. The following actions are required to model the behaviour of the `mult` box:

```
mult_init, mult_exe_exe, mult_exe_sup, mult_exe, mult_sup, mult   :: temporal.
```

mult_init is the consume phase of the box execution, while mult_exe_exe and mult_exe_sup represents the compute phase where the child `itermult` box is scheduled. These actions together create the execute phase action mult_exe of the `mult` box:

```
mult_exe ≡   mult_init ∨ mult_exe_exe ∨ mult_exe_sup .
```

The definitions of these three actions are shown in Figure 6.6, and they are mutually exclusive. mult_init pattern matches the input which succeeds if w1 and w2 are not empty (mCon succeeds). The internal input wires (iw1 and iw2) and input buffer (mult_inp) are given these values. The remaining `mult` components, and the scheduling is initialised (mult_st is set to Execute). Note that pc is unchanged, i.e. the box is not terminated. If the pattern matching fails, the next box (environment) executes.

mult_exe_exe and mult_exe_sup behaves as a scheduler for the `itermult` box in execute and super-step phase, respectively. mult_exe_exe is the execute-phase and is straightforward. The termination conditions for `mult` is _, i.e. when the internal output wire iw4 is not empty. In that case, the value of `iw4` is copied to the result buffer of `mult` (mult_res) by mult_exe_sup, and the box terminates.

The super-step action for `mult`, mult_sup, is almost the same as for a flat box, although all the internal variables must be explicitly set to be unchanged:

```
mult_init ≡
    $mult_st = #Runnable
  ∧ (if mCon<$w1> ∧ mCon<$w2>
     then (w1,w2,pc,epc)$ = (#None,#None,$pc,#epbmult)
       ∧ (mult_inp,mult_res)$ = (($w1,$w2),#None)
       ∧ (iw1,iw2,iw3,iw4)$ = ($w1,$w2#None,#None)
       ∧ (mult_st,itermult_res,itermult_st)$ = #(Execute,(None,None),Runnable)
     else (mult_st,mult_res,mult_inp)$ = #(Matchfail,None,(None,None))
       ∧ (pc,iw1,iw2,iw3,iw4)$ = #(penv,None,None,None,None)
       ∧ (itermult_res,itermult_st,epc)$ = #((None,None),Runnable),epbmult)

mult_exe_exe ≡
    $mult_st = #Execute ∧ itermult
  ∧ Unchanged (pc,w1,w2,iw4,mult_res,mult_inp)
  ∧ mult_st$ = S<$mult_st,$epc=#epbmult,termcond<bmult_st$>,isVal<iw4$>>

mult_exe_sup ≡
    $mult_st = #Super ∧ itermult ∧ epc$ = #epbmult
  ∧ Unchanged (w1,w2,iw1,iw2,mult_inp)
  ∧ mult_st$ = S<$mult_st,$epc=#epbmult,termcond<itermult_st$>,isVal<iw4$>>
  ∧ (if isVal<iw4$>
     then mult_res$ = iw4$ ∧ pc$ = #penv
     else Unchanged (pc,mult_res))
```

Figure 6.6: Definition of mult_init, mult_exe_exe and mult_exe_sup TLA actions

```
mult_sup ≡
    (iw1,iw2,iw3,iw4)$ = #(None,None,None,None)
    (itermult_res,itermult_st)$ = #((None,None),Runnable)
  ∧ (if assertOut<$mult_res,$w1>
     then w3$ = nwire<$mult_res,$w3>
       ∧ (mult_st,mult_res,mult_inp)$ = #(Runnable,None,(None,None))
     else Unchanged (w3,mult_res,mult_inp) ∧ mult_st$ = #Blocked)
```

The mult action, defined as

```
mult ≡
    ($s = #Execute ⟶
     (if $pc = #pmult
      then (if $mult_st ∈ #{Blocked,Matchfail,Terminated}
              then Unchanged (mult_res,mult_inp,mult_st,w1,w2,iw1,iw2,iw3,iw4)
                ∧ Unchanged(itermult_res,itermult_st,epc) ∧ pc$ = #penv
            else mult_exe)
      else Unchanged (mult_res,mult_inp,mult_st,w1,w2,iw1,iw2,iw3,iw4)
        ∧ Unchanged(itermult_res,itermult_st,epc)))
  ∧ ($s = #Super ⟶ mult_sup),
```

```
init      ::   temporal
init      ≡    $(s,pc,w1,w2,w3) = #(Execute,pmult,None,None,None)
          ∧    $(mult_res,mult_inp,mult_st) = #( mnnm None,(None,None),Runnable)
          ∧    $(iw1,iw2,iw3,iw4) = #(None,None,None,None)
          ∧    $(itermult_res,itermult_st,epc) = #((None,None),Runnable,epbmult)

program   ::   temporal
program   ≡    init
          ∧    □[S' ∧ env ∧ mult]_(s,pc,w1,w2,w3,mult_res,mult_inp,mult_st,
                                   iw1,iw2,iw3,iw4,itermult_res,itermult_st,epc)
```

Figure 6.7: TLA definition of the initial state init*, and the full program* program

first performs a case-split on the scheduler s. In the execute phase this is followed by a test if it is the box's turn to execute, that is if pc is pmult. If this is not the case, the box cannot be executed, and all box components (including the input wires) are left unchanged. If it holds, then the box state (mult_st) is used to check if the box can be executed, and if it holds mult_exe is executed. If it fails, all components are unchanged. Note that the pc check has to be separated from the mult_st check since in the latter case pc is updated as well. The initial state and full program are shown in Figure 6.7, and have obvious definitions following Section 6.3.

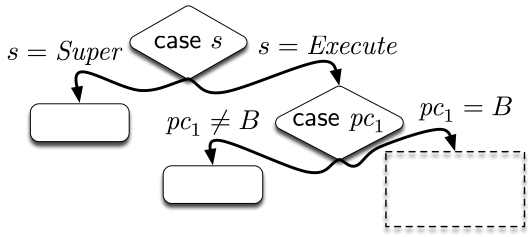
6.5 Verifying Hierarchical Hume invariants

In Hierarchical Hume, the type of the scheduler s is the same as a box state, thus requiring the proof of

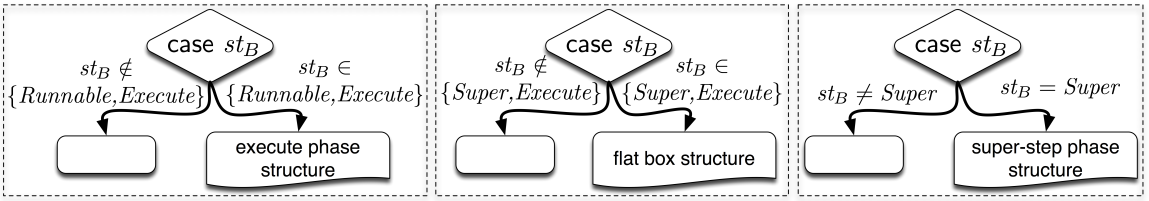
$$\Box(s \in \{Execute, Super\}). \tag{6.2}$$

This is proved in Appendix B.2, for the formal embedding. However, due the shallow mechanisation, the meta-theorem cannot be proved in Isabelle/HHume. Hence, it must be proved for each Isabelle/HHume program. To enable reuse of the tactics developed in Section 4.5, flat boxes are represented by a one-step execution phase, where $\mathit{flat}(i) \wedge i \in BS_1$ implies $\Box(st_i \notin \{Execute, Super\})$. Thus, the proofs of flat boxes are similar to those in Section 4.5. Henceforth, the focus is on invariants inside a nesting box, i.e. *nested invariant*, and *partial correctness of nesting boxes*. These are discussed separately below. Note that it assumed that all nested boxes are flat (depth 2 hierarchies).

6.5.1 Nested invariant



a) overall step case structure



b) w is in $Iiws_B$ c) w is in $IIws_B$ or $P\big(p(res_i)\big)$ d) w is in $Iows_B$

Figure 6.8: Proof plan of nested invariants

Let w be a second level (nested) wire, nested by the first level box $B \in BS_1$. Moreover, let P be the predicate to prove, box $i \in children(B)$ be the box that writes to w, and p be the projection of the element of res_i that writes to w. Thus,

$$\Box\big(w \neq \bot \Rightarrow P(w)\big) \text{ and } \Box\Big(p(res_i) \neq \bot \Rightarrow P\big(p(res_i)\big)\Big)$$

formalises the two possible properties addressed. The overall proof plan for these invariants is shown in Figure 6.9, with the premise strengthened by (6.2). Thus, only the step case differs from the proof structures previously discussed. Figure 6.8.a shows the generic structure, where empty boxes are considered trivial since w and res_i are not manipulated. The $pc_1 = B$ case results in a case split on st_B, and there are three different cases depending on the location of w: Figure 6.8.b shows the case when w is an internal input wire, where the *Runnable* and *Execute* cases are handled similarly to the execute phase of a flat box; Figure 6.8.c shows the case when w is not directly connected to B, where it is verified as a normal flat Hume box; Figure 6.8.d shows the case when w is an internal output wire, where it is verified as the super-step phase of a flat Hume box. If the invariant is over the result buffer, that is of the form $P\big(p(res_i)\big)$, then it is verified as a flat Hume invariant (as shown in Figure 6.8.c).

Hierarchical Hume enables specification of properties with respect to the input

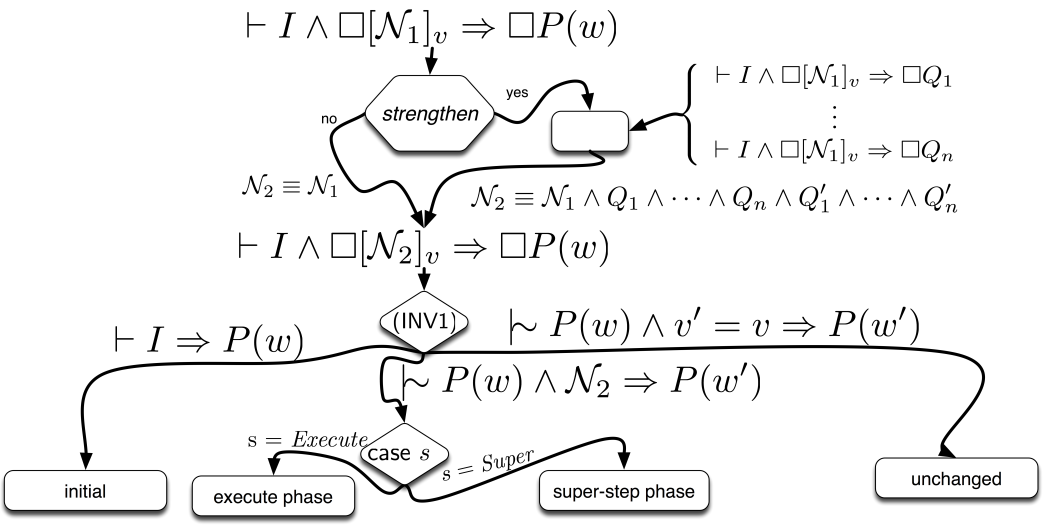


Figure 6.9: Overall proof plan of an invariant (duplicate of Figure 5.3)

buffer. These properties are of the form

$$\Box(w \neq \bot \wedge q(inp_B) \neq \bot \Rightarrow Q(w, q(inp_B)))$$

where q is a projection of one or more values of the input buffer of the parent box B[2]. In all the examples of this thesis, $*$ has not been used in the pattern of a nesting box. In these cases it can be proved that $\Box(w \neq \bot \Rightarrow q(inp_i) \neq \bot)$, which reduces the property to:

$$\Box(w \neq \bot \Rightarrow Q(w, q(inp_i))).$$

The embedding of Hierarchical Hume implies that in all the states above, except $st_B = Runnable$, inp_B is left unchanged. When $st_B = Runnable$, the input wires are copied to inp_B. These properties are verified as above, albeit with additional strengthening. This strengthening is required since the input buffer is not used by the internal wires: the same values are duplicated by the internal input wires. However, these may be consumed, thus cannot be used to specify properties as discussed in Chapter 5.

The required properties are invariants on wires from the internal input wires until w is "reached". These properties must be found in a similar way to Dijkstra's weakest precondition predicate transformer *wp* [56] in Hoare logic [70, 95]: the weakest invariants on the iws_i wires (input wires of box i that writes to w) to prove $Q(w, q(inp_i))$ is

[2]If q projects more than a single value, $q(inp_B) \neq \bot$ must be written as a tuple, that is of the form $q(inp_B) \neq \langle \bot, \cdots, \bot \rangle$. However, for simplicity, this is ignored.

first verified. Then the weakest invariants required to prove these invariants are found, and this process continues until the internal input wires $Iiws_B$ are reached. Lamport has generalised the weakest precondition predicate transformer *wp* to a weakest invariant predicate transformer *win* [119] for concurrent systems. *win* is closely related to the approach taken here, and this connection is elaborated upon in Section 10.3. Note, that this "weakest invariant approach" is manually applied.

6.5.2 Partial correctness of nesting boxes

Partial correctness properties for a box $B \in BS_1$ and $nested(B)$, is formalised as:

$$\Box\Big(p_1(inp_B) \neq \bot \wedge p_2(res_B) \neq \bot \Rightarrow P(p_1(inp_B)) \Rightarrow Q\big(p_1(inp_B), p_2(res_B)\big)\Big),$$

where p_1 and p_2 are projection functions which may be the identity function, P is the pre-condition, and Q is the post-condition. As above, if $*$ is not used in the pattern of B, this can be strengthen to

$$\Box\Big(p_2(res_B) \neq \bot \Rightarrow P\big(p_1(inp_B)\big) \Rightarrow Q\big(p_1(inp_B), p_2(res_B)\big)\Big).$$

By the definition of Hierarchical Hume, on termination of B, $Iows_B$ is copied to res_B. Since (the liveness property) termination is ignored, i.e. partial and not total correctness is verified, the proof of partial correctness is reduced to:

$$\Box\Big(p_2(Iows_B) \neq \bot \Rightarrow P\big(p_1(inp_B)\big) \Rightarrow Q\big(p_1(inp_B), p_2(Iows_B)\big)\Big),$$

which is the same type of property as discussed above, and is thus verified using the "weakest invariant approach". Note that res_B and thus $Iows_B$ can be tuples of elements/wires, and only single-wire properties have been discussed. For all examples, this has not been a problem, and Q can be split into a conjunction $Q_1 \wedge \cdots \wedge Q_n$ where all conjuncts are predicates over a single wire (and $p_1(inp_B)$).

6.6 Verifying Hierarchical Hume transformations

Figure 6.10 illustrates the flat- to nested-box transformations. Note that the same input (inp_A) and result (res_A) buffers are updated by both the source box `A` and the result box `B`. Figure 6.11 shows that the program structure is unchanged by the transformation, and `A` and `B` are similarly wired.

As defined in Section 5.4, $Trans(prog_1, prog_2)$ expresses a correct transformation.

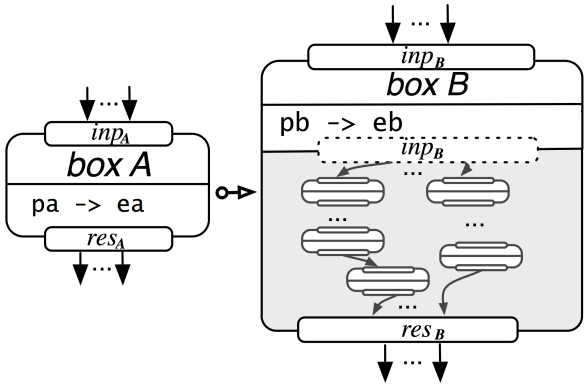


Figure 6.10: Flat to nested box transformation overview

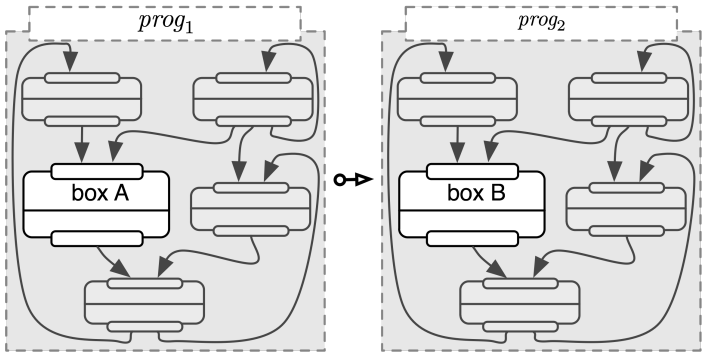


Figure 6.11: Illustration of box to box transformation of a program

Unfolding *Trans*, $prog_1$ and $prog_2$, and applying (E2), reduces the conjecture to

$$(I_2 \wedge \Box[\mathcal{N}_2]_w) \Rightarrow (\exists\, st, s, pc, con, expr.\ I_1 \wedge \Box[\mathcal{N}_1]_v). \tag{6.3}$$

The key to the proof is the following partial correctness property:

$$\vdash prog_2 \Rightarrow \Box\big(res_A \neq \bot \Rightarrow res_A = ea(inp_A)\big). \tag{6.4}$$

Further, let

$$\mathcal{N}_{2'} \triangleq \mathcal{N}_2 \wedge \big(res_A \neq \bot \Rightarrow res_A = ea(inp_A)\big) \wedge \big(res'_A \neq \bot \Rightarrow res'_A = ea(inp'_A)\big). \tag{6.5}$$

By applying the invariant strengthening discussed in Chapter 5 and (6.4), it can be shown that $\Box[\mathcal{N}_{2'}]_w \equiv \Box[\mathcal{N}_2]_w$. To prove (6.3), the refinement mapping (witnesses) for all the $\exists$-bound variables must first be found. These witnesses can be defined using $prog_2$'s variables, and, with the exception of st_A, all the mappings are the corresponding variables of $prog_2$. Now st_B ($prog_2$), alternates between *Super* and *Execute* in the

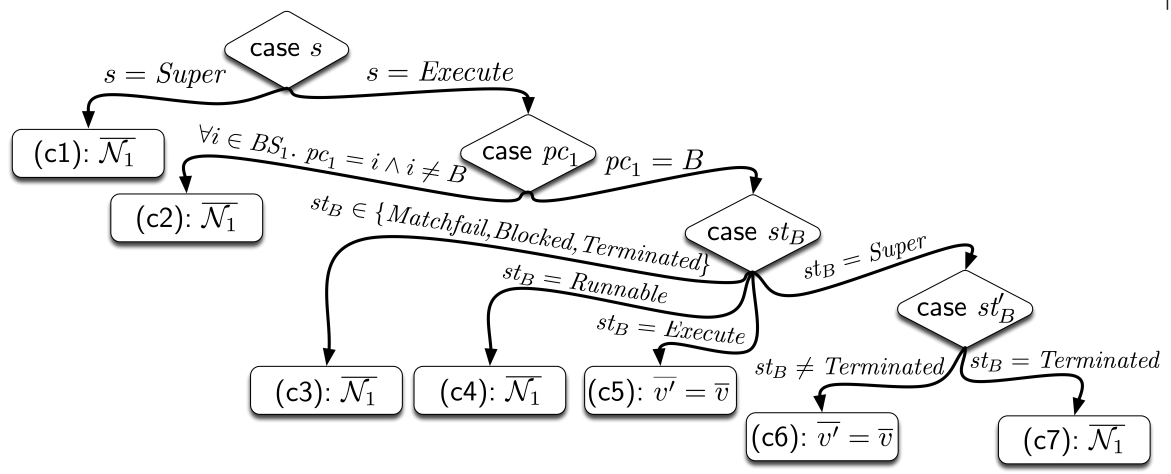


Figure 6.12: Transformation proof plan of next state action

compute phase of the box execution, while in $prog_1$, this phase is a single step where st_A is always *Execute*. Thus, the refinement mapping is defined as

$$\overline{F} \triangleq [(\textbf{if } st_B = Super \textbf{ then } Execute \textbf{ else } st_B)/st_A]F. \tag{6.6}$$

$\overline{F}$ distributes over all connectives, thus using $\Box[\mathcal{N}_{2'}]_w \equiv \Box[\mathcal{N}_2]_w$, (6.3) reduces to

$$I_2 \wedge \Box[\mathcal{N}_{2'}]_w \Rightarrow \overline{I_1} \wedge \Box[\overline{\mathcal{N}_1}]_{\overline{v}}$$

Following the derivation shown in Section 5.4, the proof reduces to the following three subgoals: (G1): $I_2 \Rightarrow \overline{I_1}$; (G2): $w' = w \Rightarrow \overline{v'} = \overline{v}$; and (G3): $\mathcal{N}_{2'} \Rightarrow [\overline{\mathcal{N}_1}]_{\overline{v}}$. Now, since initially $st_B = Runnable$ and all buffers are empty, while w contains v, and thus contains $\overline{v}$, (G1) and (G2) are trivial to show. Thus, only case (G3) is interesting and discussed further. By the definition of $[\cdots]_{\cdots}$, in the proof of (G3) it must be shown that $\mathcal{N}_{2'}$ either implies $\overline{\mathcal{N}_1}$ or it implies $\overline{v'} = \overline{v}$. Informally, there are 4 cases: (1) in the consume step the pattern matching and input copying must be the same; in the compute step there are two cases: (2) when the box terminates it must behave like $\overline{\mathcal{N}_1}$, i.e. the same values must be copied to the output buffer; (3) when the box does not terminate, this must be a stuttering step of $prog_1$, i.e. $\overline{v'} = \overline{v}$; and (4) the super-step must be the same. Figure 6.12 shows the formal proof structure of (G3). It starts by a case-analysis on s, followed by case-analysis on pc_1, st_B and st'_B in the relevant branches. This results in seven cases (c1) to (c7):

(c1) : $s = Super \wedge \mathcal{N}_2 \Rightarrow \overline{\mathcal{N}_1}$.

This should be trivial to prove since it should be unchanged between `A` and `B`.

(c2) : $s = Execute \wedge pc_1 \neq B \wedge \mathcal{N}_2 \Rightarrow \overline{\mathcal{N}_1}$.

Firstly, remember that A is the identifier of the source box, and B is the identifier of the target box. It is assumed that $A = B$ with respect to $\prec$. That is $(A \prec C) \equiv (B \prec C)$ and $(C \prec A) \equiv (C \prec B)$ for all C. The case where $pc_1 \neq B$ is trivial since only A is transformed and all other boxes are left unchanged. Note that this case involves all boxes $i \in BS_1 - \{B\}$.

(c3) : $s = \mathit{Execute} \wedge pc_1 = B \wedge st_B \in \{\mathit{Matchfail}, \mathit{Blocked}, \mathit{Terminated}\} \wedge \mathcal{N}_{2'} \Rightarrow \overline{\mathcal{N}_1}$.
The case where $pc_1 = B$, results in a case-analysis on the box state st_B. This case is the trivial case where the box is not executed.

(c4) : $s = \mathit{Execute} \wedge pc_1 = B \wedge st_B = \mathit{Runnable} \wedge \mathcal{N}_{2'} \Rightarrow \overline{\mathcal{N}_1}$.
The consume step of `A` and `B` must be the same. The main part of this case is to show that $pm(pb, iws_B) = pm(pa, iws_B)$ and $pattcopy(pb, iws_B) = pattcopy(pa, iws_B)$.

(c5) : $s = \mathit{Execute} \wedge pc_1 = B \wedge st_B = \mathit{Execute} \wedge \mathcal{N}_{2'} \Rightarrow \overline{v'} = \overline{v}$.
When $st_B = \mathit{Execute}$ then $\overline{\mathcal{N}_1}$ will execute, while $\mathcal{N}_2$ will schedule the children boxes. However, with the exception of st_B, none of the variables in v are updated, and based on the definitions of Hierarchical Hume, $st'_B = \mathit{Execute}$ or $st'_B = \mathit{Super}$. Thus, from the refinement mapping (6.6), this can be seen as a stuttering step in $prog_1$, since $\overline{v}$ is unchanged.

(c6) : $s = \mathit{Execute} \wedge pc_1 = B \wedge st_B = \mathit{Super} \wedge st'_B \neq \mathit{Terminated} \wedge \mathcal{N}_{2'} \Rightarrow \overline{v'} = \overline{v}$.
When $st_B = \mathit{Super}$, `B` either terminates or continues scheduling the children. This requires a case-split on the termination condition: $st'_B = \mathit{Terminated}$. The latter case, where $st'_B \neq \mathit{Terminated}$ is thus a stuttering step of $prog_1$. st_B is set to *Execute*, and the remaining variables in v are unchanged, thus (6.6) implies that $\overline{v}$ is unchanged.

(c7) : $s = \mathit{Execute} \wedge pc_1 = B \wedge st_B = \mathit{Super} \wedge st'_B \neq \mathit{Terminated} \wedge \mathcal{N}_{2'} \Rightarrow \overline{\mathcal{N}_1}$.
The step where `B` terminates simulates the execute step of $\overline{\mathcal{N}_1}$. The key of the proof is to show that the result buffer is updated with the same value, and the box state is the same. The latter trivially follows from the Hierarchical Hume definition and (6.6). The key here is to show that $res'_A = ea(inp_A)$. (6.5) implies that $res'_A = ea(inp'_A)$ and the definition of Hierarchical Hume implies $inp'_A = inp_A$.

6.7 Isabelle/HHume tactics

The following tactics implement the reasoning discussed above. All the tactics described in Chapter 5 are still used.

first_level_sch_tac rews automatically solves the $\Box(s \in \{Execute, Super\})$ invariant. rews must contain the program and scheduler definition, and the initial state. For example, in the program above, rews must contain the definitions of program, init and S'.

unf_nesting_tac invs box sup init omits all first level details, resulting in a subgoal containing the *Execute* phase of the nesting box, in which all definitions may be contained in box. Now, invs contains all the invariants to be strengthened with (by inv_strengthen_tac). $\Box(s \in \{Execute, Super\})$ must be in invs. The tactic unlifts from the temporal level and then from the Intensional level into Isabelle/HOL. The unchanged goal is attempted to be solved, and init is used to solve the initial state. In the last (main) goal, a case analysis on s is performed, resulting in six sub-goals. The use of $\Box(s \in \{Execute, Super\})$ then reduces this into two sub-goals. sup must contain the required definition for the $s = Super$ case. If the invariant is nested, this case is trivial. Let B be the box identifier. box must contain the required "first-level parts" of B. In the above example, this is the mult action (and none of the sub-actions). Now, in the last $s = Execute$ sub-goal, box is used to reduce this one sub-goal with the assumption that $pc_1 = B$ and $st_B \in \{Runnable, Execute, Super\}$. That is, the cases where the internal parts of the box are actually executed. Thus, it reduces the proof to the "stippled box case" of Figure 6.8.a.

Iiws_inv_tac st parent exe attempts to verify "*Iiws* type invariants" as shown in Figure 6.8.b. It assumes that unf_nesting_tac has been applied, and one sub-goal as described above is remaining. Let B be the parent box which contains this invariant. st must be the state-variable of the parent box, i.e. st_B. The tactic first applies a case-analysis to this. The $st_B \notin \{Runnable, Execute, Super\}$ cases are handled by assumptions resulting from unf_nesting_tac. Moreover, the $st_B \notin \{Runnable, Super\}$ cases are handled by the parent box. parent is assumed to contain all the required parent definitions, and these cases are attempted solved using parent. Thus, the $st_B = Execute$ case is the only one remaining. exe must contain the execute phase of the nested box(es) which consumes the wire(s) the invariant is over. The case is then attempted to be solved by execute_tac exe 1.

IIws_inv_tac st parent exe sup attempts to verify "*IIws* type invariants" as shown in Figure 6.8.c. The overall structure is similar to Iiws_inv_tac. Here however, there are two resulting cases: (1) $st_B = Execute$ and (2) $st_B = Super$. These are attempted to be solved by execute_tac exe 1 and superstep_tac sup 2.

lows_inv_tac st parent sup attempts to verify "*Iows* type invariants" as shown in Figure 6.8.d. The overall structure is similar to liws_inv_tac and llws_inv_tac, with the exception that $st_B = Super$ is the only remaining case. This is attempted to be solved by superstep_tac sup 1.

boxtransform_tac invs progs source dest inits st aux implements the transformation proofs structure discussed in Section 6.6. Let $A/prog_1$ be the name of the source box/program, and $B/prog_2$ the name of the destination box/program of the transformation. It solves a goal of the form $\vdash\ prog_2 \Rightarrow \overline{prog_1}$, that is, after instantiating the refinement mapping. progs must contain the definition of these programs, and the definition of the refinement mapping. The assumption, i.e. the next state action of $prog_2$ is first strengthen with invs (by inv_strengthen_tac). This list must contain both $\Box(s \in \{Execute, Super\})$ and the partial correctness property of $prog_2$ shown in (6.4). The following rule (see Section 3.6), which reduces the temporal level to the action level, is then applied:

theorem `refinement1`: **assumes:** $\vdash$ P $\longrightarrow$ Q **and** $\vdash$ [A]_f $\longrightarrow$ [B]_g
shows: $\vdash$ P $\wedge$ $\Box$[A]_f $\longrightarrow$ Q $\wedge$ $\Box$[B]_g

inits must contain the initial states of both program and is used to solve the first subgoal. The second sub-goal it applied to the rule (see Section 3.6):

theorem `refstep`:
assumes: $\vdash$ Unchanged v $\longrightarrow$ Unchanged w
and $\vdash$ P $\longrightarrow$ Q $\vee$ Unchanged w
shows: $\vdash$ [P]_v $\longrightarrow$ [Q]_w

where the first subgoal is solved using unchanged_tac. The second case follows the structure shown in Figure 6.12. This start by a case-analysis on s, which by $\Box(s \in \{Execute, Super\})$ reduces to the $s = Execute$ and $s = Super$ cases. source and dest contains all the box action related to A and B. Note that the actions of the nested boxes of B are not required. Now, the super-step phase action for A and B should be identical, hence the $s = Super$ case is solved by the simplifier, using source and dest. The case split on pc_1 is handled by splitting the if-then-else expression with this check (see for example the `mult` action above). The case where $st_B \in \{Blocked, Matchfail, Terminated\}$ is handled similiarly. The cases where $pc_1 \neq B$ and $st_B \in \{Blocked, Matchfail, Terminated\}$ are trivially handled by the simplifier, using source and dest. st contains the state variable of box B (i.e. st_B), and the proof follows by a case-analysis of st where the st $\in \{Blocked, Matchfail, Terminated\}$ is handled by the assumption. The remaining cases are implemented as shown in Figure 6.12. Note that in the st = *Super* case,

the "termination case" is handled by a case analysis on if-then-else expression (see the definition of mult_exe_sup in Figure 6.6).

first_level_sch_tac only holds for a first-level scheduler. unf_nesting_tac first strengthens the conjecture with the given invariants, and unlifts the conjecture to the HOL level. This has to be done first in a verification task, and can only be done once. These features have been included in this tactic since only two-level hierarchies are supported, and thus a first-level box is assumed. Boxes are scheduled in the same way at any level. Hence, if the strengthening and unlifting is moved into a separate tactic, which is applied first, then unf_nesting_tac can be generalised to any level. However, this may require more inputs, such as properties of the parent box, and user-interaction, through the additional application of the strenthening/unlifting tactic. liws_inv_tac, llws_inv_tac and lows_inv_tac can be applied to any level box. Finally, boxtransform_tac assumes a first level box. As with unf_nesting_tac, it should be possible to extend it to work at any level.

6.8 Summary & discussion

This chapter has motivated and informally defined Hierarchical Hume, formalised its semantics in TLA, and mechanised the semantics in Isabelle/TLA. This mechanisation is called Isabelle/HHume. Proof plans for verifying invariants, partial correctness and transformations have been described and implemented as tactics in Isabelle/HHume.

The Hierarchical Timing Language (HTL) [77] is a coordination layer with emphasis on real time which allows "foreign language" to define the computation. The computations are known as tasks, which can be structured into modes, while modes can be structured into modules, which can be nested. Hierarchical Hume deviates from HTL, since the "foreign language" is assumed to be the expression layer, thus the events,modes and modules are captured by a box. Moreover, whilst resource properties are important, the main design constraint of Hierarchical Hume is ensuring it conservatively extends flat Hume. The structure of Hiearchical Hume is also comparable to *statecharts* [93], and the simpler Hierarchical State Machines [31]. Here, the the motivation is managing the number of states, and *hierarchies of states* are introduced to structure the states into super- and sub-states. In Hierarchical Hume, boxes are structured, which define the transitions of states, i.e. *hierarchies of transitions* are introduced. However, since Hierarchical Hume uses the FSM model, it is still comparable to statecharts. In statecharts, the super-state can be classified into AND-state and XOR-state [61]. Following this taxonomy, a nesting box constitutes an AND-state, since it holds the conjunction of all the children boxes and wires, and not an exclusive

choice of them.

The invariant verification approach is a direct extension of the work in Section 4.5. The most common program transformations of statecharts are *program refactoring* [72], as they are mainly implemented as class hierarchies in an object oriented language. Here, commonalities or specialisation are moved into super-classes. Strictly speaking, a flat- to nested-box transformation is a Hierarchical Hume to Hierarchical Hume transformation, i.e. a *program refactoring*. However, it can also be seen as a functional language (flat box) to coordination language (nesting box) transformation, which is a *program migration*. Moreover, the expression layer is higher level than the coordination layer, thus creating a *program synthesis*. In particular, the correctness proofs are based on a synthesis type called *program refinement* where the lower level transformed program *implements* the upper level program.

Ignoring the overall program structure, and only focusing on the structure inside the resulting nesting box, the interplay between the two layers is strong. Changing one of the layers requires change of the other, which is distinct compared with synthesis techniques like Bird-Meertens Formalism [26] and calculational programming [99]. Thus, although the abstraction level is lowered, it may still be seen as a program migration. HTL supports mode refinement constrained by a non-increasing *worst case execution time* (WCET). This enables use of the most abstract representation when producing WCET guarantees, which are often easier to work with. In Hierarchical Hume, transformations are motivated by failed resource (e.g. WCET) costing, thus such a constraint does not make sense. Finally, note that the transformation proofs could have been verified as *bisimulations* in process algebras [18]. The next chapter contains several verification case-studies of Hierarchical Hume programs.

Chapter 7

Hierarchical Hume case studies

7.1 Introduction

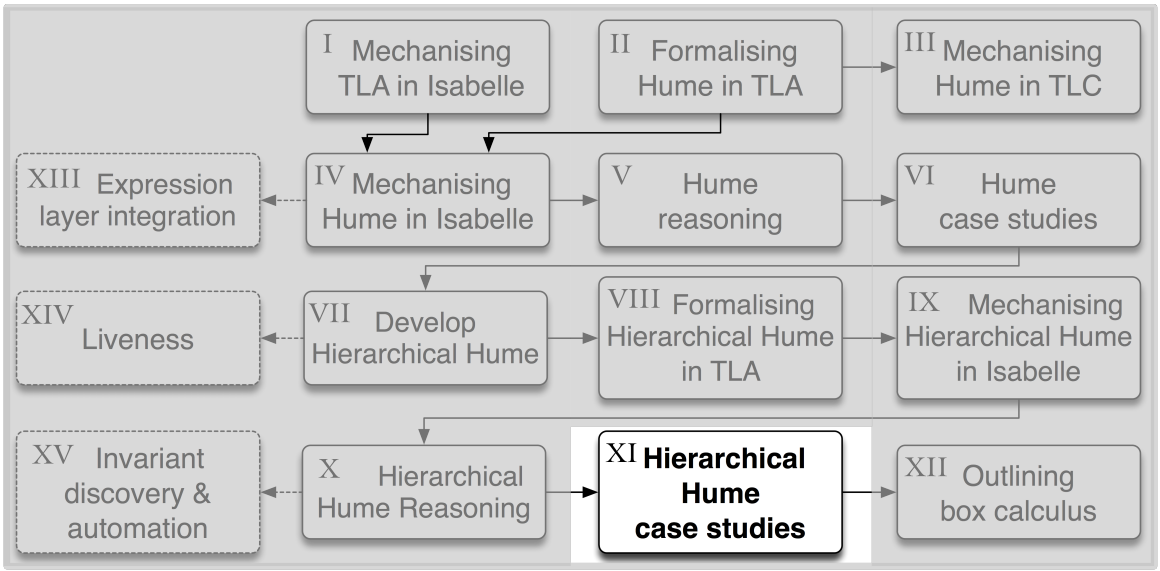
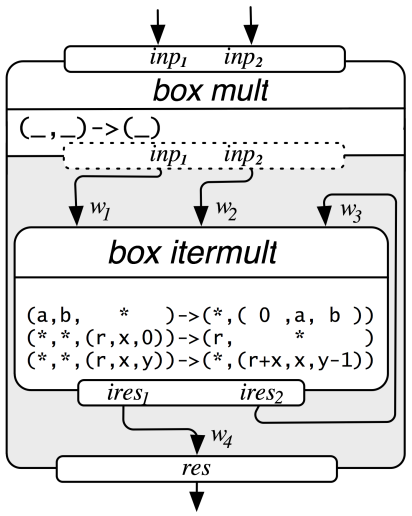


Figure 7.1: Thesis roadmap: Chapter 7

This chapter contains three case-studies in Hierarchical Hume and Isabelle/HHume, and Figure 7.1 highlights which part of the roadmap being implemented: Section 7.2 discusses property and transformation verification of a multiplication by iteration program; Section 7.3 discusses property and transformation verification of an efficient exponential function; and Section 7.4 discusses property verification of a back-pack jet propulsion system developed by NASA and originally formalised in PVS. This provides empirical evidence for both the use of Hierarchical Hume as a programming language, and the application of the verification techniques developed in Isabelle/HHume. Appendix A.5. contains the mechanised lemmas and theorems that are not listed here.

7.2 Case study HH1: multiplication by iteration

7.2.1 The program



w_1 : $\quad \Box(w_1 \neq \bot \Rightarrow w_1 = inp_1)$

w_2 : $\quad \Box(w_2 \neq \bot \Rightarrow w_2 = inp_2)$

$w_3/ires_2$:

$$\Box \left(\begin{array}{l} (w_3 \neq \bot \\ \Rightarrow \texttt{fmult}\ w_{3_1}\ w_{3_2}\ w_{3_3} = \texttt{fmult}\ 0\ inp_1\ inp_2) \wedge \\ (ires_2 \neq \bot \\ \Rightarrow \texttt{fmult}\ ires_{2_1}\ ires_{2_2}\ ires_{2_3} = \texttt{fmult}\ 0\ inp_1\ inp_2) \end{array} \right)$$

$ires_1$: $\quad \Box(ires_1 \neq \bot \Rightarrow ires_1 = \texttt{fmult}\ 0\ inp_1\ inp_2)$

w_4 : $\quad \Box(w_4 \neq \bot \Rightarrow w_4 = \texttt{fmult}\ 0\ inp_1\ inp_2)$

Figure 7.2: The `mult` box and required lemmas

The program is shown on the left side of Figure 7.2. It is an adaption of the program shown in Figure 4.3 of Chapter 4 into Hierarchical Hume, which performs multiplication as coordination iteration, represented as a feedback loop. The "interface" `mult` box now nests the `itermult` box instead of being its sibling. Thus, computation is within one cycle, instead of $N + 4$, for an iteration of depth N.

7.2.2 Partial correctness

The `mult` box performs multiplication by iteration. Thus, an important property is that the box does indeed perform multiplication. This is specified by a function `fmult` in Hume, which performs multiplication by tail-recursion:

```
fmult r _ 0 = r;
fmult r x y = fmult (r+x) x (y-1);
```

First, it is proved that fmult behaves like Isabelle/HOL multiplication $*$, when `r` is `0`:

$$\texttt{fmult}\ 0\ x\ y = x * y \tag{7.1}$$

Proof outline. This requires a generalisation of the original conjecture to

$$\forall r, x.\ \texttt{fmult}\ r\ x\ y = r + (x * y),$$

which is verified by induction on y. This is mechanised as fmult_mult_l, while (7.1), which follows directly from this, is mechanised as fmult_mult:

lemma `fmult_mult_l:` $\forall$ r x. fmult r x y = r + (x*y)
theorem `fmult_mult:` fmult 0 x y = x*y .

$\therefore$

Thus, to show partial correctness it is sufficient to show the following property

$$\Box(res \neq \bot \Rightarrow res = \texttt{fmult}\ 0\ inp_1\ inp_2) \tag{7.2}$$

The program is mechanised as `program2` in Isabelle/HHume, where the input and output streams are handled by the environment, and (7.2) is mechanised as pr2_main, and the 'scheduler property' is mechanised as p2sch The required lemmas on the different wires and the `itermult` output buffer are listed on the right hand side of Figure 7.2, and are mechanised as lemmas pr2_2 to pr2_7:

lemma `p2sch:` $\vdash$ program2 $\longrightarrow$ $\Box$($s $\in$ #{Execute,Super})
lemma `pr2_1:` $\vdash$ program2 $\longrightarrow$ $\Box$($mult_st $\in$ #{Execute,Super,Terminated}
$\longrightarrow$ isVal<$fst<mult_inp>> $\wedge$ isVal<$snd<mult_inp>>)
lemma `pr2_2:` $\vdash$ program2 $\longrightarrow$ $\Box$(isVal<$iw1 > $\longrightarrow$ @iw1 = @fst<mult_inp>)
lemma `pr2_3:` $\vdash$ program2 $\longrightarrow$ $\Box$(isVal<$iw2> $\longrightarrow$ @iw2 = @snd<mult_inp>)
lemma `pr2_4:` $\vdash$ program2 $\longrightarrow$ $\Box$(isVal<$iw1> $\wedge$ isVal<$iw2>
$\longrightarrow$ fmult<# 0, @iw1, @iw2> = fmult<# 0,@fst<mult_inp>,@snd<mult_inp>>)
lemma `pr2_5:` $\vdash$ program2 $\longrightarrow$
$\Box$((isVal<$iw3> $\longrightarrow$ fmult<fst3<@iw3>, snd3<@iw3>, thd3<@iw3>>
= fmult<# 0,@fst<mult_inp¿,@snd<mult_inp>>)
$\wedge$ (isVal<snd<$bmult_res>> $\longrightarrow$ fmult<fst3<@snd<bmult_res>>,
snd3<@snd<bmult_res>>, thd3<@snd<bmult_res>>>
= fmult<# 0,@fst<mult_inp>,@snd<mult_inp>>))
lemma `pr2_6:` $\vdash$ program2 $\longrightarrow$ $\Box$(isVal<fst<$bmult_res>>
$\longrightarrow$ @fst<bmult_res> = fmult<# 0,@fst<mult_inp>,@snd<mult_inp>>)
lemma `pr2_7:` $\vdash$ program2 $\longrightarrow$ $\Box$(isVal<$iw4>
$\longrightarrow$ @iw4 = fmult<# 0,@fst<mult_inp>,@snd<mult_inp>>)
theorem `pr2_main:` $\vdash$ program2 $\longrightarrow$ $\Box$(isVal<$mult_res>
$\longrightarrow$ @mult_res = fmult<# 0,@fst<mult_inp>,@snd<mult_inp>>)

(7.2) is then proved as follows:

Proof outline. First, the 'scheduler property' $\Box(s \in \{Execute, Super\})$ is verified

by the hume_sch_tac tactic. In the proof of the remaining lemmas, including (7.2), the unf_nesting_tac tactic, with strengthening of the previously proved lemmas, is first applied. This reduces the proof to locally "inside" the nesting `mult` box. The w_1 and w_2 wire invariants are proved by the liws_tac tactic. The $w_3/ires_2$ property, that is the loop invariant, is a (D3) type invariant (Figure 5.2), and is mechanised by pr2_5. Furthermore, it requires a (D2) type strengthening by the properties of wires w_1 and w_2. It is verified by the llws_tac tactic, followed by an application of:

$$y \neq 0 \wedge \texttt{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ a\ b \Rightarrow \texttt{fmult}\ (r + x)\ x\ (y - 1) = \texttt{fmult}\ 0\ a\ b,$$

mechanised by

$$\textbf{lemma}\ \texttt{fmult3:}\quad [\![\ \mathsf{y} \neq 0;\ \mathsf{fmult\ r\ x\ y} = \mathsf{fmult\ 0\ a\ b}\ ]\!] \Longrightarrow \mathsf{fmult\ (r+x)\ x\ (y\text{-}(Suc\ 0))} = \mathsf{fmult\ 0\ a\ b}$$

which follows directly by the definition of `fmult`. $ires_1$ is verified by the llws_tac tactic, followed by the rule:

$$y = 0 \wedge \texttt{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ a\ b \Rightarrow r = \texttt{fmult}\ 0\ a\ b,$$

mechanised by:

$$\textbf{lemma}\ \texttt{fmult4:}\quad [\![\ \mathsf{y} = 0;\ \mathsf{fmult\ r\ x\ y} = \mathsf{fmult\ 0\ a\ b}\ ]\!] \Longrightarrow \mathsf{r} = \mathsf{fmult\ 0\ a\ b}$$

which also follows directly by the definition of `fmult`. Finally, by using these lemmas, w_4 and (7.2) are verified by the lows_tac tactic. $\therefore$

7.2.3 Transformation proof

Let `emult` be a box which directly applies the `fmult` function on the input as shown in Figure 7.3. The transformation from the `emult` box into the `mult` box, as shown in the same figure, is an example of the well-known recursion to iteration transformation [83]. As discussed in Section 6.6, the proof of this transformation mainly reduces to the proof of (7.2). To verify the transformation, the refinement mapping must be defined. Let mult_st be the mechanisation of the state of the `mult` box. The refinement mapping mst is then mechanised as[1]:

[1]For simplicity, only the state variable is bound.

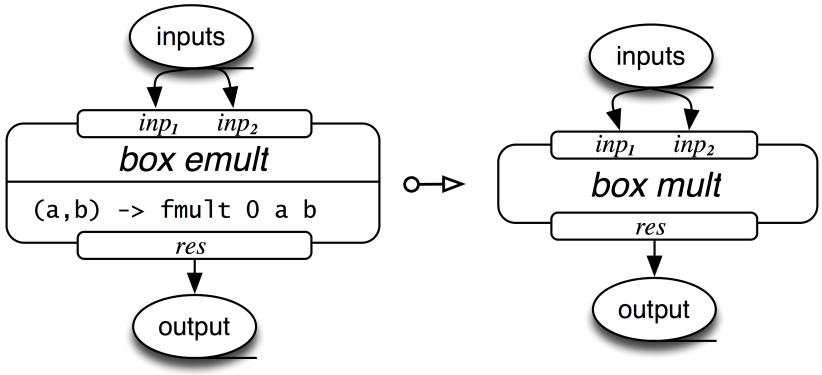


Figure 7.3: Multiplication recursion to iteration transformation

mst ≡ if mult_st = #Super then #Execute else mult_st

Remember that the ∃ operator is written ∃∃ in Isabelle/TLA, and accept a function from a state function to a temporal formula. Let prog1 be the specification of the source program of the transformation shown in Figure 7.3, as a function over the state variable of the `emult` box. Moreover, let program1 be `prog1` with the state variable bound by ∃∃. The transformation is thus verified by the following conjectures:

lemma `main_trans_lemma:` ⊢ program2 ⟶ prog1 mst
lemma `main_trans_lemma2:` ⊢ program2 ⟶ (∃∃ st. prog1 st)
theorem `trans_thm:` ⊢ program2 ⟶ program1.

Proof outline. Lemma main_trans_lemma is verified mainly by the boxtransform_tac tactic using (7.2) and the 'scheduling property'. Lemma main_trans_lemma2 follows from Lemma main_trans_lemma by applying rule (E1), while Theorem trans_thm follows from Lemma main_trans_lemma2 by folding the definition program1. ∴

7.3 Case study HH2: an efficient exponential box

7.3.1 The program

The second case-study is an effective implementation of the exponential function. The program is a porting of imperative code used as an assignment in a module by Andrew Ireland at Heriot-Watt University, originally taken from [81, page 36]. The left hand side of Figure 7.4 shows this imperative code.

The right hand side of Figure 7.4 shows the Hierarchical Hume implementation of the same function. Note that due to the static nature of wires, duplication is required.

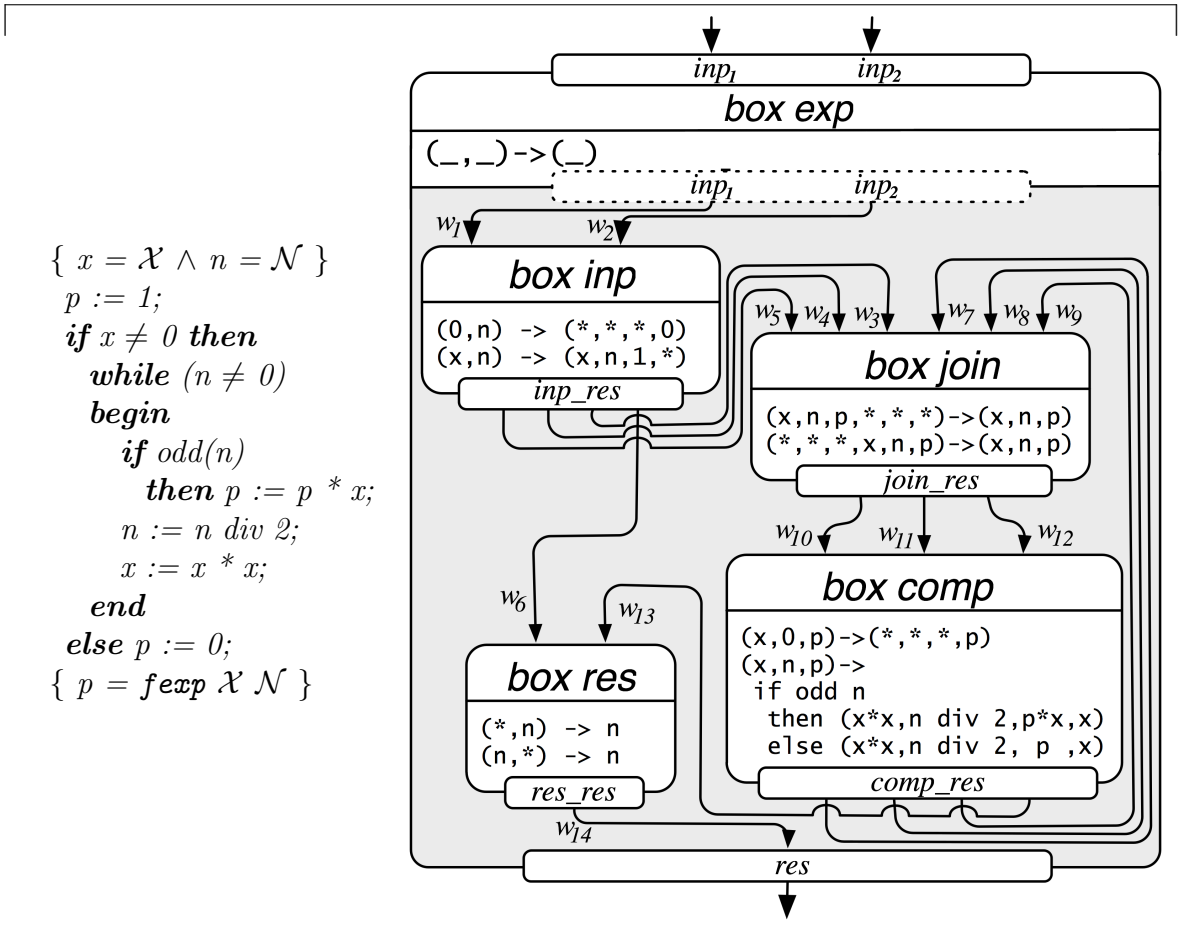


Figure 7.4: Left: the exponential function in imperative code. Right: the **exp** *box implementing the exponential function in Hierarchical Hume*

The `inp` box implements the first ***if*** statement, while the ***while*** loop is split into the `join` and `comp` boxes. Again, due to the static nature of wires, the `join` box is introduced due to the wire duplication, and to simplify the `comp` box. Finally, the `res` box joins the two branches of the first ***if*** statement. The comp box uses the `odd` function, defined recursively in Hume:

```
odd 0 = false;   odd 1 = true;   odd n = odd (n-2);
```

7.3.2 Partial correctness

The imperative code of Figure 7.4 specifies the partial correctness property by Hoare triples. It is specified using the following function, which implements the exponential operator recursively:

```
fexp x 0 = (if x = 0 then 0 else 1);
fexp x n = (if x = 0 then 0 else x*(fexp x (n-1)));
```

Note that the undefined `fexp 0 0` (0^0) case is assumed to be `0`. The partial correctness property is formalised as:

$$\Box(res = \texttt{fexp}\ inp_1\ inp_2), \tag{7.3}$$

and relies on the following facts:

$$p*(\texttt{fexp}\ x\ n) = \texttt{fexp}\ a\ b \wedge \texttt{odd}\ n \Rightarrow (p*x)*(\texttt{fexp}\ (x*x)\ (n\ \mathsf{div}\ 2)) = \texttt{fexp}\ a\ b \tag{7.4}$$

$$p*(\texttt{fexp}\ x\ n) = \texttt{fexp}\ a\ b \wedge \neg(\texttt{odd}\ n) \Rightarrow p*(\texttt{fexp}\ (x*x)\ (n\ \mathsf{div}\ 2)) = \texttt{fexp}\ a\ b \tag{7.5}$$

Proof outline. (7.4) and (7.5) are mechanised as odd1 and nodd1, and are proved by first generalising to

$$\begin{array}{rcl} \texttt{odd}\ n & \Rightarrow & \texttt{fexp}\ x\ n = x * \big(\texttt{fexp}\ (x*x)\ (n\ \mathsf{div}\ 2)\big) \\ \neg(\texttt{odd}\ n) & \Rightarrow & \texttt{fexp}\ x\ n = \texttt{fexp}\ (x*x)\ (n\ \mathsf{div}\ 2), \end{array}$$

which are mechanised as fexp2 and fexp3, and are proved by a two-step induction on n, with 0 and 1 as base cases. $\therefore$

The lemmas of the `fexp` and `odd` functions are mechanised as follows:

```
lemma fexp1:  fexp 0 n = 0
lemma fexp2:  odd n ⟶ fexp x n = x*(fexp (x*x) (n div 2))
lemma fexp3:  ¬(odd n) ⟶ fexp x n = fexp (x*x) (n div 2)
lemma odd1:   ⟦ p*(fexp x n) = fexp a b; odd n ⟧
                  ⟹ (p*x)*(fexp (x*x) (n div 2)) = fexp a b
lemma nodd1:  ⟦ p*(fexp x n) = fexp a b; ¬odd n ⟧
                  ⟹ p*(fexp (x*x) (n div 2)) = fexp a b.
```

The partial correctness property (7.3) is mechanised as exp_main, and is verified as follows:

Proof outline. The 'scheduling property' is first verified by the hume_sch_tac tactic. In all the remaining lemmas the unf_nesting_tac tactic is first applied, strengthened by the scheduling property and the previously verified lemmas. The properties are mechanised (in the correct order) below. For all (D1) and (D2) style invariants, the property is first verified for the result buffer, then the (output) wire(s). Henceforth, to ease the reading, the result buffer is normally ignored in the proof outlines, albeit these properties are mechanised still.

Now, the property that w_1 equals inp_1 and w_2 equals inp_2 when not empty, is verified by the liws_tac tactic. From the definition of `inp`, w_6 will always be 0, and this is only the case when inp_1 is 0, thus

$$\Box(w_6 \neq \bot \Rightarrow w_6 = \texttt{fexp}\ inp_1\ inp_2) \tag{7.6}$$

which follows from the definition of `fexp`. If not empty, w_5 and w_4 wires equal inp_1 and inp_2, while w_3 will be 1. This is verified by the llws_tac tactic.

The ***while*** loop in the imperative version is represented by the `join` and `comp` boxes, connected by the w_7, w_8, w_9 and w_{10}, w_{11}, w_{12} wires. Thus, together these form a (D4) type invarant, and in the imperative program the loop invariant is $p{*}\texttt{fexp}\ x\ n = \texttt{fexp}\ \mathcal{X}\ \mathcal{N}$. In Hierarchical Hume, the loop has a (D2) dependency on the w_3, w_4 and w_5 wires, which is the entry of the loop. Thus,

$$\Box(w_3 \neq \bot \wedge w_4 \neq \bot \wedge w_5 \neq \bot \Rightarrow w_5 * (\texttt{fexp}\ w_3\ w_4) = (\texttt{fexp}\ inp_1\ inp_2),$$

is first verified by the llws_tac tactic. Now, in the Hierarchical Hume loop, p, x and n are represented by the `join` and `comp` result buffers and the w_7, w_8, w_9 and w_{10}, w_{11}, w_{12} wires. Moreover, while in the imperative case it is implicit that $x \neq 0$, this must be captured by the Hierarchical Hume invariant, which becomes:

$$\begin{aligned}
\Box(\ & (join_res_1 \neq \bot \wedge join_res_2 \neq \bot \wedge join_res_3 \neq \bot \\
& \quad \Rightarrow join_res_3 * (\texttt{fexp}\ join_res_1\ join_res_2) = (\texttt{fexp}\ inp_1\ inp_2) \wedge join_res_1 \neq 0) \\
& \wedge (w_7 \neq \bot \wedge w_8 \neq \bot \wedge w_9 \neq \bot \Rightarrow w_9 * (\texttt{fexp}\ w_7\ w_8) = (\texttt{fexp}\ inp_1\ inp_2) \wedge w_7 \neq 0) \\
& \wedge (comp_res_1 \neq \bot \wedge comp_res_2 \neq \bot \wedge comp_res_3 \neq \bot \\
& \quad \Rightarrow comp_res_3 * (\texttt{fexp}\ comp_res_1\ comp_res_2) = (\texttt{fexp}\ inp_1\ inp_2) \wedge comp_res_1 \neq 0) \\
& \wedge (w_{10} \neq \bot \wedge w_{11} \neq \bot \wedge w_{12} \neq \bot \\
& \quad \Rightarrow w_{12} * (\texttt{fexp}\ w_{10}\ w_{11}) = (\texttt{fexp}\ inp_1\ inp_2) \wedge w_{10} \neq 0)).
\end{aligned}$$

This is mechanised by expl10 and verified by the llws_tac tactic, followed by the application of (7.4) and (7.5). Then, the 'exit step' of the loop induces

$$\Box(w_{13} \neq \bot \Rightarrow w_{13} = \texttt{fexp}\ inp_1\ inp_2),$$

verified by the llws_tac tactic. With this, (7.6), and the definition of `res`,

$$\Box(w_{14} \neq \bot \Rightarrow w_{14} = \texttt{fexp}\ inp_1\ inp_2).$$

is verified by the lows_tac tactic, from which (7.3) directly follows. $\therefore$

The partial correctness lemmas and theorem are mechanised as follows:

lemma pr_sch: ⊢ program ⟶ □($s ∈ #{Super,Execute})
lemma exp_l1: ⊢ program ⟶ □(isVal<$w1> ⟶ @w1 = @fst<exp_inp>)
lemma exp_l2: ⊢ program ⟶ □(isVal<$w2> ⟶ @w2 = @snd<exp_inp>)
lemma exp_l3a: ⊢ program ⟶ □(isVal<fst4<$inp_res>>
⟶ @fst4<inp_res> = @fst<exp_inp> ∧ @fst4<inp_res> ≠ # 0)
lemma exp_l3: ⊢ program ⟶ □(isVal<$w3>
⟶ @w3 = @fst<exp_inp> ∧ @w3 ≠ # 0)
lemma exp_l4a: ⊢ program ⟶ □(isVal<snd4<$inp_res>>
⟶ @snd4<inp_res> = @snd<exp_inp>)
lemma exp_l4: ⊢ program ⟶ □(isVal<$w4>
⟶ @w4 = @snd<exp_inp>)
lemma exp_l5a: ⊢ program ⟶ □(isVal<thd4<$inp_res>>
⟶ @thd4<inp_res> = # 1)
lemma exp_l5: ⊢ program ⟶ □(isVal<$w5> ⟶ @w5 = # 1)
lemma exp_l6a: ⊢ program ⟶ □(isVal<for4<$inp_res>>
⟶ @for4<inp_res> = # 0)
lemma exp_l6: ⊢ program ⟶ □(isVal<$w6> ⟶ @w6 = # 0)
lemma exp_l7a: ⊢ program ⟶ □(isVal<for4<$inp_res>>
⟶ @fst<exp_inp> = # 0)
lemma exp_l7: ⊢ program ⟶ □(isVal<$w6> ⟶ @fst<exp_inp> = # 0)
lemma exp_l8: ⊢ program ⟶ □(isVal<$w6>
⟶ @w6 = fexp<@fst<exp_inp>,@snd<exp_inp>>)
lemma exp_l9a: ⊢ program
⟶ □(isVal<fst4<$inp_res>> ∧ isVal<snd4<$inp_res>> ∧ isVal<thd4<$inp_res>>
⟶ (@thd4<inp_res>*fexp<@fst4<inp_res>,@snd4<inp_res>>)
= fexp<@fst<exp_inp>,@snd<exp_inp>>)
lemma exp_l9: ⊢ program ⟶ □(isVal<$w3> ∧ isVal<$w4> ∧ isVal<$w5>
⟶ (@w5*fexp<@w3,@w4>) = fexp<@fst<exp_inp>,@snd<exp_inp>>)
lemma expl10_val1: ⊢ program
⟶ □((isVal<fst4<$comp_res>>)=(isVal<snd4<$comp_res>>)
∧ (isVal<fst4<$comp_res>>)=(isVal<thd4<$comp_res>>))
lemma expl10_val2: ⊢ program
⟶ □((isVal<fst3<$join_res>>)=(isVal<snd3<$join_res>>)
∧ (isVal<fst3<$join_res>>)=(isVal<thd3<$join_res>>))

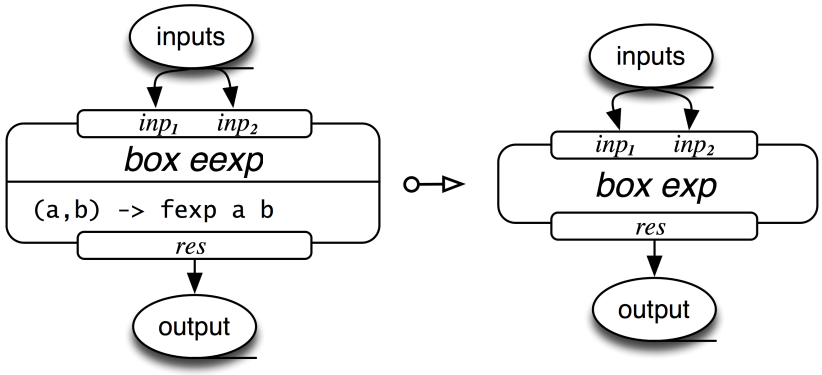


Figure 7.5: Exponential function transformation

lemma `expl10:` ⊢ program ⟶
□((isVal<fst3<$join_res>> ∧ isVal<snd3<$join_res>> ∧ isVal<thd3<$join_res>> ⟶
(@thd3<join_res>*fexp<@fst3<join_res>,@snd3<join_res>>)
= fexp<@fst<exp_inp>,@snd<exp_inp>> ∧ @fst3<join_res> ≠ # 0)
∧ (isVal<$w10> ∧ isVal<$w11> ∧ isVal<$w12> ⟶
(@w12*fexp<@w10,@w11>) = fexp<@fst<exp_inp>,@snd<exp_inp>>
∧ @w10 ≠ # 0)
∧ (isVal<fst4<$comp_res>> ∧ isVal<snd4<$comp_res>> ∧ isVal<thd4<$comp_res>>
⟶ (@thd4<comp_res>*fexp<@fst4<comp_res>,@snd4<comp_res>>)
= fexp<@fst<exp_inp>,@snd<exp_inp>> ∧ @fst4<comp_res> ≠ # 0)
∧ (isVal<$w7> ∧ isVal<$w8> ∧ isVal<$w9> ⟶ (@w9*fexp<@w7,@w8>)
= fexp<@fst<exp_inp>,@snd<exp_inp>> ∧ @w7 ≠ # 0))

lemma `expl11a:` ⊢ program ⟶ □(isVal<for4<$comp_res>>
⟶ @for4<comp_res> = fexp<@fst<exp_inp>,@snd<exp_inp>>)

lemma `expl11:` ⊢ program ⟶ □(isVal<$w13> ⟶
@w13 = fexp<@fst<exp_inp>,@snd<exp_inp>>)

lemma `expl12a:` ⊢ program ⟶ □(isVal<$res_res> ⟶
@res_res = fexp<@fst<exp_inp>,@snd<exp_inp>>)

lemma `expl12:` ⊢ program ⟶ □(isVal<$w14> ⟶
@w14 = fexp<@fst<exp_inp>,@snd<exp_inp>>)

theorem `exp_main:` ⊢ program ⟶ □(isVal<$exp_res> ⟶
@exp_res = fexp<@fst<exp_inp>,@snd<exp_inp>>)

7.3.3 Transformation

The transformation is shown in Figure 7.5. By using (7.3), it follows the exact same structure as the proof of Theorem trans_thm in Section 7.2. The following lemmas and

theorems are mechanised for this proof:

lemma `main_trans_lemma:` $\vdash$ program $\longrightarrow$ eprog mst
lemma `main_trans_lemma:` $\vdash$ program $\longrightarrow$ ($\exists\exists$ st. eprog st)
theorem `trans_thm:` $\vdash$ program $\longrightarrow$ eprogram

7.4 Case study HH3: the SAFER system

7.4.1 Overview of the SAFER system

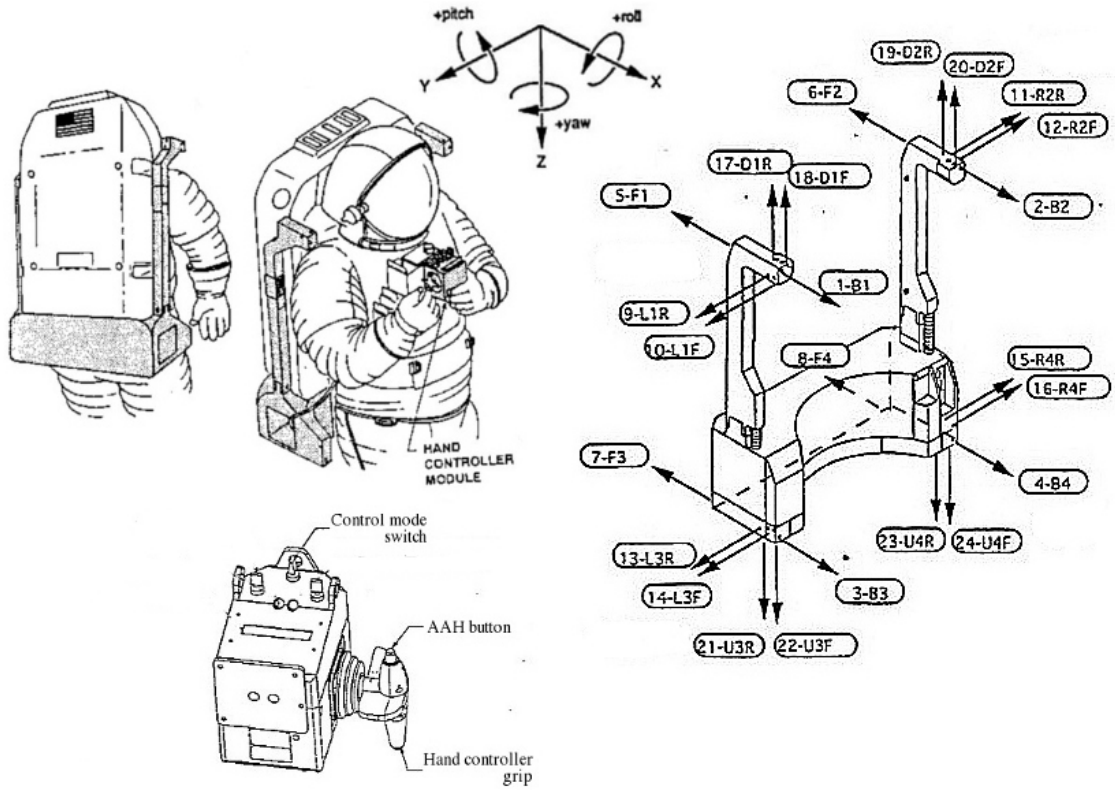


Figure 7.6: Left-to-right: SAFER deployed on an astronaut; the six degree of movement axis; the mounting of the thrusters. Bottom: the hand controller (source: [153])

Simplified Aid For Eva Rescue (SAFER) is a lightweight backpack propulsion system developed by NASA [153]. It is intended to provide self-rescue capabilities for astronauts outside the spacecraft in space. The left-most drawing of Figure 7.6 shows the system deployed on an astronaut. It allows movement along and around all three axes, as illustrated in the middle diagram. This is controlled by a hand-controller operated by the astronaut – and by an *automatic attitude hold* (AAH), which attempts to nullify rotation.

The hand controller is shown on the bottom-left hand side of Figure 7.6. In this embedding, only the *control mode switch*, *hand controller grip* and *AAH button* are used: the control mode switch changes the mode between rotation and transition; the hand controller grip has three rotary axes and one transverse axis. To issue a command, the astronaut moves the hand controller grip from the null centre position to the mechanical hardstop on the axis. To terminate a command, the grip is either released or moved back to the null centre position. In the *transition* mode, a (potentially compound) command may institute manoeuvring in the $\pm X$, $\pm Y$, $\pm Z$ and $\pm$ *pitch* directions; while in the $\pm X$, $\pm$*pitch*, $\pm$*roll* or $\pm$*yaw* directions in *rotation* mode. The AAH button (on the hand controller grip) switches the AAH on (one-click) and off (two-clicks).

Commands from the hand-controller and various sensors are sent to the software, which returns a list of the thrusters that should be fired. A thruster is filled with pressured GN_2 gas. If the vent of a thruster is open, this gas is released, creating a force and acceleration in the opposite direction. The SAFER system consists of twenty four thrusters, organised such that four thrusters are pointing in each of the $\pm X$, $\pm Y$ and $\pm Z$ directions. This enables a six-degree of freedom manoeuvring control along and around the three axes. The right hand side of Figure 7.6 shows the mounting of the thrusters and thruster names, which are adopted in the Hume embedding. The software system is embedded in Hume.

Requirements

The SAFER system is a medium-sized real world example, with a large set of requirements which should be verified. The system has been mechanised [12, 153], and several of the requirements have been independently verified [12, 153, 194]. Most of the requirements are listed in Appendix C of [153], while [12] and [194] suggest some additional properties. Here, the following requirements are treated:

1. when AAH is inactive, and no hand grip commands are present, there should be no thruster firings (source: [194]);

2. at most, one translation command shall be acted upon, with the axes chosen in priority order X, Y, Z (source: [153], property 40);

3. hand controller rotation commands shall suppress any translation commands that are present, but AAH-generated rotation commands may coexist with translations (source: [153], property 39);

4. no two selected thrusters should oppose one another, i.e. have cancelling effect with respect to the centre of mass (source: [12] and Section C.4.1 of [153]);

7.4.2 The Flying Scotsman: SAFER in Hierarchical Hume

Overview of the embedding

This section discusses the embedding of parts of the software system of SAFER in Hierarchical Hume. The system was originally mechanised in PVS [164] in [153], and has later been embedded in VDM-SL [130] in [12]. The main functionality of the software system can be divided into fault detection and EVA/ground checkout and manoeuvring control. Both the PVS and VDM-SL versions only focus on the manoeuvring control, and this is followed here as well. Moreover, the software system accepts inputs from the hand controller and sensors, and produces a list of thrusters to execute. In PVS, the complete hand controller and all sensors are included, albeit only the relevant parts for the AAH and thruster selection are actually defined. VDM-SL is a testing tool requiring that the specification is executable. Consequently, all the undefined PVS functions are stripped out. This is also the case in Hume, thus the VDM-SL embedding is the closest to the one described here. The system inputs are from the hand controller and the output is to the thrusters. In addition, the system must store the state of the AAH.

In both PVS and VDM-SL, SAFER is modelled as a state machine: each cycle reads the sensors and the current AAH state, and produces a list of thrusters to fire and a new AAH state. One such cycle is assumed to last $5ms$. Each part of the system is represented as a function, and in each cycle, the computation order is controlled by function composition. In Hume, the parts are represented by boxes, and the computation order is controlled by the wiring. To achieve a one-to-one mapping between a SAFER cycle and Hume cycle, the program is nested by a box `SAFER`, shown in Figure 7.7. This program can be divided into three parts: the hand controller, which consists of the `grip_command` box; the AAH which comprises all boxes prefixed by `AAH`, and the thruster selection logic, which comprises the remaining boxes. These parts are discussed separately below, preceded by a discussion of the types. The full source code is listed in Appendix C, where the hand-controller inputs are represented by a (dummy) generator, and the thrusters fired are converted to a string and sent to standard output.

Types

Figure 7.8 shows all the type definitions. A thruster is either on or off, i.e. there is no speed. `axis_command` is used to show the direction for one axis, while `command` contains the `axis_command` for all three axes (translation or rotation). In both PVS

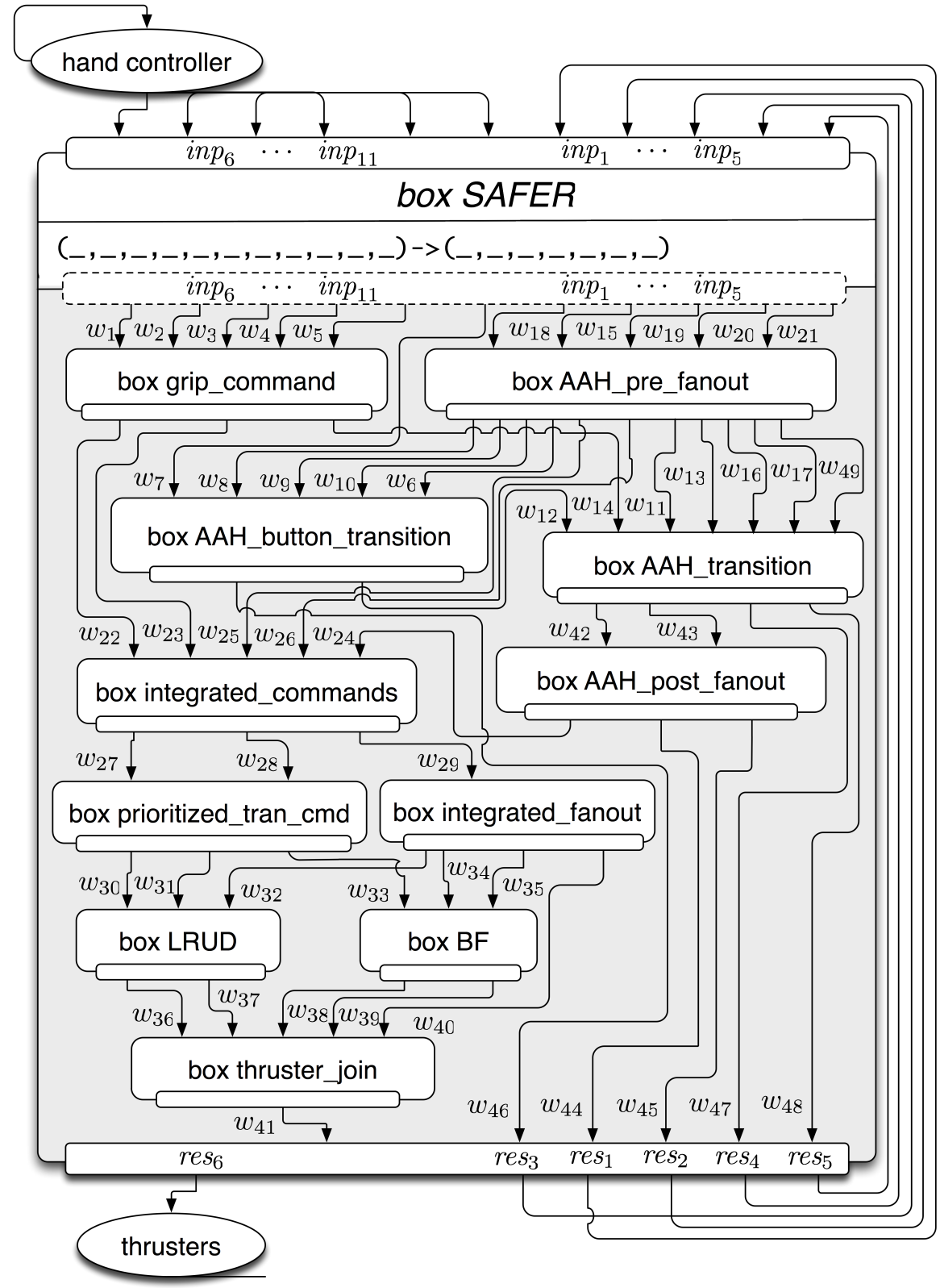


Figure 7.7: The `SAFER` *box: the SAFER system in Hierarchical Hume*

```
data axis_command = NEG | ZERO | POS;
type command = (axis_command,axis_command,axis_command);
type axis_pred = (bool,bool,bool);

data thruster_name = B1 | B2 | B3 | B4 | F1 | F2 | F3 | F4 | L1R | L1F
     | R2R | R2F | L3R | L3F | R4R | R4F | D1R | D1F | D2R | D2F | U3R
     | U3F | U4R | U4F;
type thruster_list = [thruster_name];

data control_mode_switch = ROT | TRAN;
data AAH_control_button = button_up | button_down;

data AAH_engage_state = AAH_off | AAH_started | AAH_on | pressed_once
     | AAH_closing | pressed_twice;
```

Figure 7.8: SAFER types in Hume

and VDM-SL, function types over each axis are heavily used. Hume does not allow function types on wires. However, since the functions are over the axes, this can be represented as a triple, as illustrated by `command` and `axis_pred`. `thruster_name` defines the names of each of the 24 thrusters, and follows the naming scheme from Figure 7.6. `control_mode_switch` is the mode read from the hand-controller, while `AAH_control_button` is the current value of the AAH button on the hand-controller. Finally, `AAH_engage_state` is used to model the AAH button state.

The hand controller

With the exception of the AAH control button, the `grip_command` box handles all the inputs from the hand controller. The hand controller can be moved among four axes, and the control mode switch is either set to rotation (`ROT`) or transition (`TRAN`). The `grip_command` box turns these signals into a transition command and two duplicated rotation commands. The rotation command is duplicated since it is required by both the thruster selection logic and the AAH:

```
box grip_command
  in  (vert,horiz,trans,twist::axis_command, mode :: control_mode_switch)
  out (tran,rot1,rot2::command)
match
  (v,h,tr,tw,TRAN,_) -> ((h,tr,v),(ZERO,tw,ZERO),(ZERO,tw,ZERO))
| (v,h,tr,tw,ROT,_)  -> ((h,ZERO,ZERO),(v,tw,tr),(v,tw,tr));
```

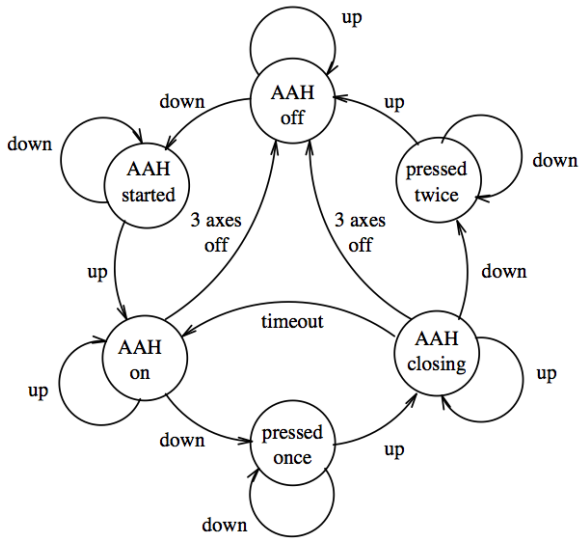


Figure 7.9: SAFER AAH engagement state diagram (source: [153])

The automatic attitude hold (AAH)

The AAH capability is invoked to maintain a near-zero rotation rate. Thus, the AAH only creates a rotation command. This requires sensor input, in addition to hand controller input and the state in the previous cycle, which are not provided in either the PVS or the VDM-SL version. In PVS, the command is not generated, while a dummy value is created in VDM-SL. The latter approach is also followed here. The AAH contains many other interesting requirements, thus the remaining parts of the AAH are embedded. The AAH is represented by the following four boxes: `AAH_pre_fanout`, `AAH_post_fanout`, `AAH_button_transition` and `AAH_transition`. The `AAH_post_fanout` generates a dummy `(ZERO,ZERO,ZERO)` command, which represents the rotation command from the AAH. With the exception of this, `AAH_pre_fanout` and `AAH_post_fanout` simply "fans out" the inputs to several other boxes, and are thus not discussed further.

The AAH has five state components, which must be remembered between each cycle, forming the feedback loops of box `SAFER`. These are the *active axis* (type `axis_pred`) which identifies which axis the AAH is currently active. Since the AAH will only generate a rotation command, this refers to (*pitch, roll, yaw*). *ignore hcm* (type `axis_pred`) identifies on which axis the hand controller (*hcm*) can be ignored. *toggle* keeps track of the status of the AAH, e.g. if it is engaging or disengaging. Figure 7.9 shows how it is updated. *time-out* is used to separate two single clicks from a double-click. And, the *clock* is a simple counter incremented in each cycle. The AAH engagement diagram of Figure 7.9 is implemented by the `AAH_button_transition` box:

```
box AAH_button_transition ..
  (AAH_off    ,button_down,_,_,_)   -> (AAH_started,AAH_started)
| (AAH_started,button_down,_,_,_)   -> (AAH_started,AAH_started) ...
```

The box accepts the current "engage value", the value of the AAH button, the active axis, the clock and time-out values, and returns the new "engage value". This value is remembered for the next cycle, and is used by the `AAH_transition` box. The result is thus duplicated. The remaining AAH state components are updated by the `AAH_transition` box:

```
box AAH_transition ...
  (     _    ,AAH_off,AAH_started ,(r1,r2,r3),    _     ,cl,tout)
     -> ((true,true,true),(r1 != ZERO,r2 != ZERO,r3 != ZERO),cl+1,tout)
| (     _    ,   _   ,  AAH_off   ,    _     ,   ihcm   ,cl,tout)
     -> ((false,false,false),ihcm,cl+1,tout)
| ((a1,a2,a3),AAH_on ,pressed_once,(r1,r2,r3),(i1,i2,i3),cl,tout)
     -> ((naa a1 r1 i1, naa a2 r2 i2, naa a3 r3 i3),
         (i1,i2,i3),cl+1,cl+click_timeout)
| ((a1,a2,a3),  eng  ,     _      ,(r1,r2,r3),(i1,i2,i3),cl,tout)
     -> ((naa a1 r1 i1,naa a2 r2 i2,naa a3 r3 i3),(i1,i2,i3),cl+1,tout);
```

The inputs of the box are: the (old) active axis; the new and old engage value; the rotation command from the hand controller (that is `grip_command`); the (old) ignore hcm; the (old) clock; and the old time-out. It produces the new active axis, ignore hcm, clock and time-out.

The ignore hcm will only change value when the AAH is starting (engage is changing from `AAH_off` to `AAH_started`). It holds for an axis if the rotation command (from the hand controller) is `ZERO` for that axis, and can thus be ignored; the clock is always incremented by 1, i.e. one cycle tick; the time-out is only changed if the button is pressed (once) while the AAH is on. This time-out "counter" is then initialised to see if the button is pressed once more within $0.5s$. This is the definition of the two-clicks of the AAH control button, and the AAH is then switched off (by `AAH_button_transition`). Here, `click_timeout` is a constant, which is 100 since 100 cycles of $5ms$ is $0.5s$; the active axis holds for all axes when the AAH is starting (engage is changing from `AAH_off` to `AAH_started`), and is `false` for all axes when AAH is disengaged. For all other cases, it depends on the current active axis value for the given axis, together with the rotation command and ignore hcm value for the same axis. The `naa` function returns the new active axis value for an axis, when given these parameters:

```
naa true ZERO  _   = true;
naa true  _   true = true;
naa  _    _    _   = false;
```

Thruster selection

The thruster selection logic has two functionalities. Firstly, the commands from the hand controller and AAH are joined to create one (translation and rotation) command. Here, a hand-controller rotation command takes precedence over a translation command, while translation command can co-exist with rotation commands from the AAH. Moreover, a translation can only be acted upon in one axis at a time, with the priority X, Y, Z. Secondly, based on this command, the correct set of thrusters are selected. The first part is achieved by the `integrated_commands`, `integrated_fanout` and `prioritized_tran_cmd` boxes. The `integrated_commands` box is defined as follows:

```
box integrated_commands ..
  (tran,(ZERO,ZERO,ZERO),    _     ,(false,false,false),    _     )
    -> (tran,(ZERO,ZERO,ZERO),true)
| ( _  ,( r  , p  , ya ),    _     ,(false,false,false),    _     )
    -> ((ZERO,ZERO,ZERO),(r,p,ya),false)
| (tran,(ZERO,ZERO,ZERO),(a1,a2,a3),         _         ,    _     )
    -> (tran,(a1,a2,a3),true)
| ( _  ,( r  , p  , ya ),(a1,a2,a3),         _         ,(i1,i2,i3))
    -> ((ZERO,ZERO,ZERO), (comb_rot_cmds r a1 i1, comb_rot_cmds p a2 i2,
          comb_rot_cmds ya a3 i3), false);
```

The box inputs are: the translation and rotation commands from the `grip_command`; the (dummy) AAH rotation command; the (old) active axis; and the old ignore hcm. It returns a translation command and a combined rotation command, together with a boolean used by the `prioritized_tran_cmd` box. This holds if a non-`ZERO` translation command is created.

Now, if there is no rotation command from the hand controller and AAH is off for all axes, then there is only the translation command. If there is a rotation command and AAH is off for all axes, then the rotation command is used and no translation command is generated. If there is no rotation command from the hand controller, and AAH is on for at least one axis, then the translation command and rotation command from the AAH is used. Finally, if rotation command is present and AAH is active (for at least one axis), then there is no transition command, and the rotation commands from the AAH and hand controller are combined. This is achieved by applying the `comb_rot_cmds` function on each axis:

```
comb_rot_cmds  ZERO aah  _   = aah;
comb_rot_cmds   _   aah true = aah;
comb_rot_cmds  hcm   _   _   = hcm;
```

The function accepts, for one axis, the rotation command from the hand controller and AAH, together with the ignore hcm for that axis. If there is no command from the grip then the AAH value is used. If this is not the case, and the ignore hcm holds (meaning ignore the hand controller), then the AAH command is used. If not, then the hand controller has priority. The transition command and boolean value from the `integrated_commands` box are sent to `prioritized_tran_cmd` box:

```
box prioritized_tran_cmd ...
  (      tran      ,false) ->       tran
| ((ZERO,ZERO,ZERO),  _  ) -> (ZERO,ZERO,ZERO)
| ((ZERO,ZERO,zacc),  _  ) -> (ZERO,ZERO,zacc)
| ((ZERO,yacc, _  ),  _  ) -> (ZERO,yacc,ZERO)
| ((xacc, _  , _  ),  _  ) -> (xacc,ZERO,ZERO);
```

A `false` boolean denotes ignore the translation command; the remaining matches implements the X,Y, Z priority. The `integrated_fanout` box "fans out" the rotation command.

The second part of the thruster selection logic selects which thrusters to fire based on the integrated command. The left, right, up and down (LRUD) thrusters result from Y, Z and *roll* commands, while back and forward (BF) thrusters result from the X, *pitch* and *yaw* commands. Based on this classification, each type of thruster is found by two boxes: the `BF` box fires the `B1-B4` and `F1-F4` `thruster_name` from Figure 7.8, while the `LRUD` box fires the remaining. The boxes are implemented by pattern matching, each containing 27 (3^3) matches, since each command may have 3 different values (and there are 3 commands):

```
box LRUD ...
   | (NEG,NEG,POS)   -> ([],[])
   | (NEG,ZERO,NEG)  -> ([L1R],[L1F,L3F])
...
box BF ...
   | (NEG,NEG,POS)   -> ([B3],[B1,B4])
   | (NEG,ZERO,NEG)  -> ([B2,B4],[])
...
```

`LRUD` and `BF` returns two lists of thrusters, a list of mandatory thrusters and a list of optional thrusters. These 4 lists are joined into a single list by the `thruster_join` box. The concatenation depends on the rotation command:

```
box thruster_join ..
  (lm,lo,bm,bo,(ZERO,ZERO,ZERO)) -> (lm ++ lo ++ bm ++ bo)
| (lm,lo,bm,_ ,(ZERO,ZERO, _  )) -> (lm ++ lo ++ bm)
| (lm,_ ,bm,bo,( _  , _  ,ZERO)) -> (lm ++ bm ++ bo)
| (lm,_ ,bm,_ ,( _  , _  , _  )) -> (lm ++ bm );
```

The ++ operator combines two lists. In addition to the four lists of thrusters, it accepts the rotation command. In all cases, the mandatory lists are fired. If this is a `(ZERO,ZERO,ZERO)` command, the thrusters in both optional lists are also fired; if the *pitch* and *yaw* commands are empty only the optional LRUF thrusters are fired; if not, then if the *roll* command is `ZERO` then only the optional BF thrusters are fired; finally, if none of these holds then only the two mandatory thruster lists are joined.

7.4.3 Verification of SAFER requirements

The program is mechanised in Isabelle/HHume as previously described, and the hand controller and thrusters are handled by the environment. Moreover, Isabelle/HOL's inductive data type, list and function package are used to represent Hume types and functions. The hand-controller and thruster are handled by the environment.

Generic properties

The four requirements are verified separately in the following sections. There are some generic properties required to verify several of these requirements. Firstly, the 'scheduler property' $\Box(s \in \{Execute, Super\})$ is verified by the hume_sch_tac tactic. Secondly, all the internal input wires, like $\Box(w_1 \neq \bot \Rightarrow w_1 = inp_6)$, are verified by the unf_nesting_tac and liws_tac tactics. All the verified lemmas, as well as the lemmas and theorems discussed below, are mechanised in Appendix A.5.1.

Requirement 1

The first requirement states that if the AAH is inactive and there are no hand grip commands, then no thrusters are fired. This property is formalised as follows:

$$\begin{aligned}\Box(&\langle inp_6, inp_7, inp_8, inp_9\rangle = \langle \texttt{ZERO}, \texttt{ZERO}, \texttt{ZERO}, \texttt{ZERO}\rangle \\ &\wedge inp_1 = \langle \texttt{false}, \texttt{false}, \texttt{false}\rangle \Rightarrow res_6 \neq \bot \Rightarrow res_6 = \langle\rangle)\end{aligned} \tag{7.7}$$

inp_6 to inp_9 are the inputs from the hand controller grip, meaning it is in a neutral position, while inp_1 is the active axis of the AAH. Since they are all `false`, the AAH is disabled for all axes. Note that this property is independent of the dummy `(ZERO,ZERO,ZERO)` command from the AAH. (7.7) is then verified as follows:

Proof outline. In all proofs, the unf_nesting_tac tactic is first applied, strengthened by the previously verified properties. Moreover, all the lemmas have the same assumptions as (7.7). First, the llws_tac tactic is used to verify that `grip_command` produces `ZERO`-transition and rotation commands on w_{22} and w_{23}. This, together with the $inp_1 =$

$\langle$`false`, `false`, `false`$\rangle$ assumption, implies that the transition w_{27} and rotation w_{29} commands from `integrated_commands` are `ZERO`-commands. This is used to show that the wires w_{30} to w_{35} are all `ZERO`. From these facts, it is shown that both the mandatory and optional thruster lists from `BF` and `LRUD` on wires w_{36} to w_{39} are empty `[]`. All of these properties are verified by the llws_tac tactic. Since all the input thruster lists to `thruster_join` are empty, an empty list is generated on wire w_{41}. On termination, this value is copied to res_6, thus (7.7) holds. The properties on w_{41} and res_6 are verified by the lows_tac tactic. ∴

Requirement 2

The second requirement consists of two parts. The first part states that, at most, one translation command shall be acted upon. Wires w_{33}, w_{30} and w_{31} contain "the final" X,Y and Z transition, before they are turned into thruster lists by `LRUD` and `BF`. Thus, if all of them are full, then at least two of them must be `ZERO`. This property is formalised by creating a predicate max_one_axis over three `axis_command`s in Isabelle/HOL, which holds if, and only if, at least two of them are `ZERO`. This first part is thus formalised as follows:

$$\Box\big(w_{33} \neq \bot \wedge w_{30} \neq \bot \wedge w_{32} \neq \bot \Rightarrow \mathsf{max\_one\_axis}(w_{33}, w_{30}, w_{31})\big) \tag{7.8}$$

Proof outline. If the input on w_{28} is `false`, then the command (triple) on w_{27} is "fanned out" to the three output wires. For this case, it is first verified that when w_{28} has the non-empty `false` value, then, if not empty, w_{27} is $\langle$`false`, `false`, `false`$\rangle$, and (7.8) trivially holds. If the input on w_{28} is `true`, (7.8) follows from the definition of the box (pattern matches) and max_one_axis. Both of these cases are verified by the llws_tac tactic, and the definition of max_one_axis. ∴

The second part of this requirement asserts that the transition command has an X, Y, Z priority. Again, this is formalised over the w_{33}, w_{30} and w_{31} wires. Now, the input from the hand-grip controller for the X and Y axis are inp_7 and inp_8. The requirement is then formalised over these definitions as follows:

$$\Box(w_{30} \neq \bot \wedge w_{30} \neq \mathtt{ZERO} \Rightarrow inp_7 = \mathtt{ZERO}) \tag{7.9}$$

$$\Box(w_{31} \neq \bot \wedge w_{31} \neq \mathtt{ZERO} \Rightarrow inp_7 = \mathtt{ZERO} \wedge inp_8 = \mathtt{ZERO}). \tag{7.10}$$

The first part verified that only one of w_{33}, w_{30} and w_{31} can be non-`ZERO`. The w_{30} wire is the Y axis generated translation command. (7.9) states that if w_{30} is a non-`ZERO` command, then the X input is `ZERO`, thus X has higher priority than Y. The w_{31} wire

is the Z axis generated translation command, and (7.10) asserts that if the w_{30} wire contains a non-`ZERO` command, then the X and Y inputs are `ZERO`, thus Z has lower priority than X and Y. These two properties show the X, Y, Z priority.

Proof outline. The proofs of (7.9) and (7.10) are by case analysis on the control mode (inp_{10} = `ROT` or inp_{10} = `TRAN`), and all properties are verified by the llws_tac tactic. In the inp_{10} = `ROT` case, only an X transition command can be generated from `grip_command`. Thus, the w_{22} wire is $\langle inp_7, \texttt{ZERO}, \texttt{ZERO}\rangle$. In the inp_{10} = `TRAN` case, the same wire is $\langle inp_7, inp_8, inp_6\rangle$. `integrated_commands` either generates the same translation command on w_{27} as w_{22} or a `ZERO`-command. In the second case, w_{28} is `false`, whilst it is `true` in the first case. In the case where w_{28} is `false`, w_{30} and w_{31} will be `ZERO`, thus (7.9) and (7.10) hold. Moreover, w_{30} and w_{31} will, if not `ZERO`, be given the second and third values of w_{27} respectively. In the inp_{10} = `ROT`, these are always `ZERO`. The inp_{10} = `TRAN` case holds by the logic of the `prioritized_tran_cmd`. The inp_{10} = `ROT` and inp_{10} = `TRAN` cases are verified separately, where (7.9) and (7.10) follows by the following rule (see Section 3.6), since $(inp_{10} \neq \texttt{ROT}) \equiv (inp_{10} = \texttt{TRAN})$:

lemma `inv_case`:
assumes: $\vdash \mathsf{P} \longrightarrow \Box(\mathsf{A} \longrightarrow \mathsf{B})$ **and** $\vdash \mathsf{P} \longrightarrow \Box(\neg\mathsf{A} \longrightarrow \mathsf{B})$
shows: $\vdash \mathsf{P} \longrightarrow \Box\mathsf{B}$ $\therefore$

Requirement 3

The third requirement also consists of two parts. The first part asserts that any hand controller translation commands shall be suppressed by hand controller rotation commands. The existence of a rotation command depends on the control mode of the hand controller, thus this property is formalised as follows:

$$\Box(inp_{10} = \texttt{TRAN} \wedge inp_9 \neq \texttt{ZERO} \wedge w_{27} \neq \bot \Rightarrow w_{27} = \langle \texttt{ZERO}, \texttt{ZERO}, \texttt{ZERO}\rangle) \tag{7.11}$$

$$\begin{aligned}\Box(&inp_{10} = \texttt{ROT} \wedge \langle inp_6, inp_9, inp_8\rangle \neq \langle \texttt{ZERO}, \texttt{ZERO}, \texttt{ZERO}\rangle \\ &\wedge w_{27} \neq \bot \Rightarrow w_{27} = \langle \texttt{ZERO}, \texttt{ZERO}, \texttt{ZERO}\rangle)\end{aligned} \tag{7.12}$$

Proof outline. When the control mode is `TRAN`, then only a *pitch* rotation command is possible, which is given on inp_9, while in `ROT` it is possible to rotate around all the axes. The property asserts that if the input of `SAFER` contains a rotation command, then the generated translation command on wire w_{27} (from `integrated_commands`) is a `ZERO`-command. Both these properties are verified by first showing that `grip_command` produces `ZERO` translation commands on w_{22}. This is then used to prove (7.11) and (7.12). All properties are verified by the llws_tac tactic. $\therefore$

The second part asserts that translation may co-exist with rotation commands from the AAH. This is formalised by asserting that as long as there are no rotation commands from the hand-controller, then the generated translation command is the translation command given by the inputs. Again, this is formulated as two theorems, depending on the control mode of the hand controller:

$$\Box(inp_{10} = \texttt{TRAN} \wedge inp_9 = \texttt{ZERO} \wedge w_{27} \neq \bot \Rightarrow w_{27} = \langle inp_7, inp_8, inp_6 \rangle) \tag{7.13}$$

$$\begin{aligned}\Box(&inp_{10} = \texttt{ROT} \wedge \langle inp_6, inp_9, inp_8 \rangle = \langle \texttt{ZERO}, \texttt{ZERO}, \texttt{ZERO} \rangle \\ &\wedge w_{27} \neq \bot \Rightarrow w_{27} = \langle inp_7, \texttt{ZERO}, \texttt{ZERO} \rangle)\end{aligned} \tag{7.14}$$

Proof outline. The proof of (7.13) and (7.14) follows the same pattern as the proof of (7.11) and (7.12). ∴

Requirement 4

```
tc []          =  True
tc (B1#vs)     =  (F1 ∉ set vs)                                  ∧  tc vs
tc (F1#vs)     =  (B1 ∉ set vs)                                  ∧  tc vs
tc (B2#vs)     =  (F2 ∉ set vs)                                  ∧  tc vs
tc (F2#vs)     =  (B2 ∉ set vs)                                  ∧  tc vs
tc (B3#vs)     =  (F3 ∉ set vs)                                  ∧  tc vs
tc (F3#vs)     =  (B3 ∉ set vs)                                  ∧  tc vs
tc (B4#vs)     =  (F4 ∉ set vs)                                  ∧  tc vs
tc (F4#vs)     =  (B4 ∉ set vs)                                  ∧  tc vs
tc (L1R#vs)    =  (R2R ∉ set vs)  ∧  (R2F ∉ set vs)              ∧  tc vs
tc (L1F#vs)    =  (R2R ∉ set vs)  ∧  (R2F ∉ set vs)              ∧  tc vs
tc (R2R#vs)    =  (L1R ∉ set vs)  ∧  (L1F ∉ set vs)              ∧  tc vs
tc (R2F#vs)    =  (L1R ∉ set vs)  ∧  (L1F ∉ set vs)              ∧  tc vs
tc (L3R#vs)    =  (R4R ∉ set vs)  ∧  (R4F ∉ set vs)              ∧  tc vs
tc (L3F#vs)    =  (R4R ∉ set vs)  ∧  (R4F ∉ set vs)              ∧  tc vs
tc (R4R#vs)    =  (L3R ∉ set vs)  ∧  (L3F ∉ set vs)              ∧  tc vs
tc (R4F#vs)    =  (L3R ∉ set vs)  ∧  (L3F ∉ set vs)              ∧  tc vs
tc (D1R#vs)    =  (U3R ∉ set vs)  ∧  (U3F ∉ set vs)              ∧  tc vs
tc (D1F#vs)    =  (U3R ∉ set vs)  ∧  (U3F ∉ set vs)              ∧  tc vs
tc (U3R#vs)    =  (D1R ∉ set vs)  ∧  (D1F ∉ set vs)              ∧  tc vs
tc (U3F#vs)    =  (D1R ∉ set vs)  ∧  (D1F ∉ set vs)              ∧  tc vs
tc (D2R#vs)    =  (U4R ∉ set vs)  ∧  (U4F ∉ set vs)              ∧  tc vs
tc (D2F#vs)    =  (U4R ∉ set vs)  ∧  (U4F ∉ set vs)              ∧  tc vs
tc (U4R#vs)    =  (D2R ∉ set vs)  ∧  (D2F ∉ set vs)              ∧  tc vs
tc (U4F#vs)    =  (D2R ∉ set vs)  ∧  (D2F ∉ set vs)              ∧  tc vs
```

Figure 7.10: The tc function: Thruster consistency mechanised in Isabelle/HOL

The fourth requirement asserts that no two thrusters being fired should have a cancelling effect, i.e. oppose one another. This property is called *thruster consistency* [12], and is mechanised in Isabelle/HOL as the primitive recursive tc function over a list of thrusters, shown in Figure 7.10. There, set is a built-in function which turns a list into a set. The fourth property is then formalised as follows:

$$\Box(res_6 \neq \bot \Rightarrow \mathsf{tc}\ res_6). \tag{7.15}$$

Proof outline. The proof of (7.15) uses the set BF_set $\equiv \{$B1,B2,B3,B4,F1,F2,F3,F4$\}$. Moreover, a predicate BF_Type x for a thruster x (of type `thruster_name`), defined by pattern matching, holds if x $\in$ BF_set, while BF_Types extends this to a list of thrusters. LRUD_set, LRUD_Type and LRUD_Types are similarly defined for LRUD thrusters. The following property

$$\mathsf{BF\_Types}\ bs \wedge \mathsf{LRUD\_Types}\ ls \wedge \mathsf{tc}\ bs \wedge \mathsf{tc}\ ls \Rightarrow \mathsf{tc}\ (bs \cdot ls) \tag{7.16}$$

is first verified by induction on bs. It asserts that the concatenation of two thruster consistent lists of different "types", as defined above, results in a thruster consistent list. Then, tc w_{38}, tc w_{39}, tc $(w_{38} \cdot w_{39})$ and BF_Types w_{38}, BF_Types w_{39}, BF_Types $(w_{38} \cdot w_{39})$ are verified by first applying the llws_tac tactic. The remaining sub-goals are proved by the Isabelle/HOL auto tactic, which implicitly contains the required rewrite rules generated from the definitions of the tc, BF_Type and BF_Types functions. The same is verified for w_{36} and w_{37}, although they are of LRUD_Types, thus LRUD_Type and LRUD_Types generated rewrite rules are used instead of the rules from BF_Type and BF_Types, albeit this is automatically handled by the auto tactic. These properties and (7.16), are strong enough to prove $\Box(w_{41} \neq \bot \Rightarrow \mathsf{tc}\ w_{41})$, where the llws_tac tactic is first applied, followed by an application of (7.16) and the standard Isabelle/HOL simplifier[2]. (7.15) follows directly from this using the lows_tac tactic. ∴

7.5 Summary & discussion

This chapter has shown use of Hierarchical Hume and the reasoning techniques introduced in Chapter 6. These experiments have shown that the tactics developed in Chapter 6 have a high degree of automation for provable properties, albeit they quickly become slow with the size of the program. For example, using a 2.4 GHc Intel core 2 Duo iMac with 1 GB memory, many of the SAFER lemmas could take over half an

[2]Actually, the result buffer of `thruster_join` is verified this way, and the property of w_{41} follows by the lows_tac tactic

hour to solve. This is a problem in Isabelle, since the proof is not stored, as in e.g. PVS. Thus each time Isabelle is restarted, the tactic is reapplied, meaning the already verified proofs have to be re-searched.

One possibility for optimising the tactics is to use other built-in Isabelle reasoning tools, such as the *sledgehammer* tactic, which applies automatic resolution based provers to verify a goal [142].

Another solution is a more controlled rewriting, which may require a deeper embedding. Firstly, the super-step phase, which can only consume wires and empty result buffers, is unnecessary slow, since the shallow embedding cannot capture this in general. Secondly, the unf_nesting_tac tactic reduces a proof of a nested property to within the box capturing the wires/buffers the property is over. However, it does not remove non-required assumptions like other boxes, which introduce unnecessary proof search. Moreover, the tactic specifies detailed application of rules and tactics on particular subgoals, and does not capture the overall reasoning pattern. Consequently, the tactics may fail if the program and property deviates slightly from the expectations.

Proof planning [32] is a higher-level method of reasoning, where common patterns of reasoning are captured by a proof plan. Abstracting the tactic into a proof plan would enable more re-use, since it would be less exposed to minor changes. Isabelle supports proof planning through IsaPlanner [57, 58]. More controlled rewriting can be achieved by a proof plan called *rippling* [33], which guarantees termination. Thus, it may be possible to solve the looping of the vending machine example of Chapter 5. Rippling was originally developed for the step-case of inductive proofs, and may thus be suitable for the inductive style of verifying invariants in TLA. Rippling within the Hume/TLA context is discussed further in Section 9.4.

The tactics cannot solve, or provide guidance, for non-theorems or theorems that require strengthening or generalisation. Even the simplest (D1) style wire invariant requires a proof of the result buffer, albeit this detail was abstracted over in text of some of the case studies. The other invariant styles require more strengthening.

The SAFER system is the largest Hierarchical Hume case study, and verification and testing has independently been discussed using PVS [153, 194] and VDM-SL [12]. VDM-SL is fully automatic, however, it can only check selected cases and thus cannot give formal correctness guarantees. In [194], Di Vito creates proof tactics in PVS for systems modelled as state machines. He achieves a higher degree of automation for the SAFER properties, compared to the Hierarchical Hume version. The reduced level of automation can be attributed to: *i*) the work is at the programming language level (a lower level of abstraction); *ii*) extra overhead from embedding Hierarchical Hume on top of TLA; *iii*) extra overhead from the Hierarchical Hume into Isabelle/HHume

translation; *iv*) a more complex coordination layer embedding of the computation, compared to a high-level and atomic functional representation in PVS. Finally, the Isabelle/HHume version can be extended with liveness, whilst in PVS, this will require a complete re-embedding of the system.

Note that both the invariant proofs and transformation proofs could have been verified in other state-based refinement systems, like B or Event-B, however, these cannot be extended with liveness, and integration with the expression layer, as discussed in Section 9.2, may not be so direct.

Currently, in a failed costing, the program is transformed, and the transformation is verified ad-hoc as described above. The next chapter describes a different approach where the transformation and verification are joined by outlining a box calculus for transformations.

Chapter 8

Towards a box calculus for transformations[1]

8.1 Introduction

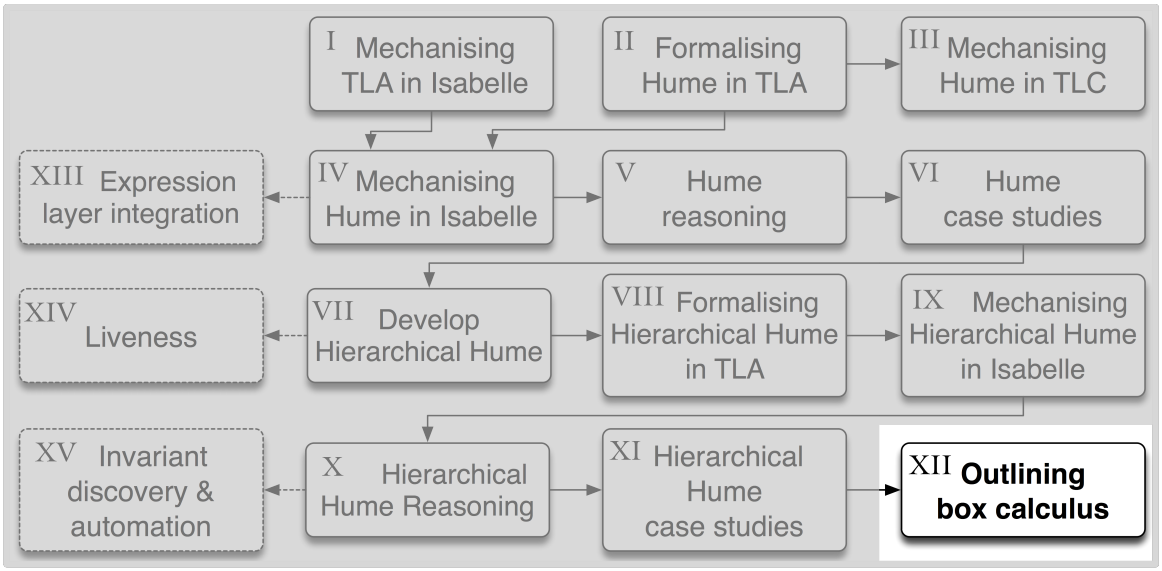


Figure 8.1: Thesis roadmap: Chapter 8

The verification of transformations in Chapter 7 was ad-hoc: a program was first transformed, and then verified. To enhance automation and user interaction, a better approach may be to define a calculus to direct these transformations. The transformed program is synthesised, and no ad-hoc verification is required, thus introducing the *correctness by construction* principle [14]. Common transformations can then be captured by *strategies* which combines lower level rules. Moreover, by defining the calculus at the Hume level, the users are liberated from the underlying logical details.

[1]The main contents of this chapter have been published in [87].

In this chapter such a calculus is proposed, and Figure 8.1 highlights which part of the roadmap is being implemented here. The calculus is intended to guarantee that each step of a transformation preserves the program's behaviour. The approach is merely outlined, and although an underlying TLA logic is assumed, the logical details are informal. The properties discussed here are not formally proved. This is beyond the scope of this thesis.

The Hume layer dependency is strong within a transformation – manipulating one layer necessarily requires manipulation of the other. Thus, in order to define the calculus, more details of the expression layer must be incorporated into the discussion. This is achieved by (informally) updating the Hume structural operational semantics (SOS) [112] with hierarchical features and using these in the TLA embedding. This is discussed in the next section, together with the syntax and semantics of the rules. This is followed by a description of rules and strategies in Section 8.3. Here, a selection of rules and strategies are derived while the remaining relevant rules and strategies are summarised. Section 8.4 provides two examples of applying the calculus, before relevant work is discussed, and the chapter is concluded.

8.2 Rule syntax and semantics

To simplify the presentation, the non-relevant features of the Hume SOS are ignored. Moreover, the calculus is intended for transformation, and not property, verification. This requires the ability to manipulate coordination features of nested components, meaning these must be hidden ($\exists$-bound). Representing them as free variables, as in Chapters 6 and 7, will impose too many restrictions on usage.

A Hierarchical Hume program configuration thus consists of a triple

$$\langle \theta, \eta, bcs \rangle :$$

θ is the wire heap with allocated space for each wire. It also holds potential initial wire values; η is the internal heap, including internal wires for hierarchical boxes. The consistent use of heap instead of stack reflects the Hume SOS. *bcs* is a list of box configurations. Each box configuration consists of the elements

$$\langle id, iws, ows, rs, ii, nci, io, ibcs \rangle :$$

id is the box's name; *iws* is a list of locations holding the input wires; *ows* is a list of locations holding the output wires; *rs* is a list of matches; *ii* is a list of locations of internal input wires; *io* is a list of locations of internal output wires; *nci* are the

non-connected internal wires, i.e. the internal wires that are not in *ii* and *io*; and *ibcs* is a list of box configurations of internal (nested) boxes. For simplicity, a Hierarchical Hume box may have more than one match, meaning the termination condition depends on which pattern succeeds. This is a natural extension of Hierarchical Hume. A flat box is represented as $\langle id, iws, ows, rs, [], [], [], [] \rangle$.

$$run_{bcs}$$

is a predicate on the pairs of before heaps $\langle \theta, \eta \rangle$ and after heaps $\langle \theta', \eta' \rangle$ for the *bcs* box configurations. It represents the execution of the program, and the full TLA program specification thus becomes

$$\exists \eta : Init_\theta \wedge Init_\eta \wedge \Box[run_{bcs}]_{\langle \theta, \eta \rangle}.$$

Other auxiliary variables like box state *st* and the scheduler *s* will probably be required but these are ignored here due to the high-level nature of the discussion. The box calculus consists of a set of conditional rewrite rules. A rule changes the triple $\langle \theta, \eta, bcs \rangle$ and has the syntax

$$\langle \theta, \eta, bcs \rangle \vdash \mathbf{Rule}(X_1, \cdots, X_n) \Downarrow \langle \theta', \eta', bcs' \rangle.$$

This should be read as "**Rule** with parameters $X_1, \cdots, X_n$ will, under the configuration $\langle \theta, \eta, bcs \rangle$ create the configuration $\langle \theta', \eta', bcs' \rangle$". To achieve a set of rules that is expressive enough, steps that change timing behaviour must be allowed. It is therefore imperative that the preconditions are strong enough to ensure that the actual behaviour remains unchanged. This is mostly a coordination issue and the nature of this layer often requires temporal properties. Thus, the HW-Hume level is used as a starting point for the calculus, and the examples in Section 8.4 are in HW-Hume.

A transformation from the configuration $\langle \theta, \eta, bcs \rangle$ into the configuration $\langle \theta', \eta', bcs' \rangle$ is represented as expected in TLA:

$$\big(\exists \eta' : Init_{\theta'} \wedge Init_{\eta'} \wedge \Box[run_{bcs'}]_{\langle \theta', \eta' \rangle}\big) \Rightarrow \big(\exists \eta : Init_\theta \wedge Init_\eta \wedge \Box[run_{bcs}]_{\langle \theta, \eta \rangle}\big).$$

For simplicity the execute phase is assumed to be atomic. This simplifies for example joining two boxes. With the given assumptions (see rule derivation in Section 8.3), this is straightforward in an atomic execute phase. In a sequential (non-atomic) phase, this is more involved since the boxes execute in different states. However, this is only allowed in nesting boxes where η is $\exists$-bound. Thus, it is still provable, but requires a more complex refinement mapping, and the proof may require auxiliary variables.

Such details are ignored henceforth. Now, to simplify reasoning, combining the TLA rules (E1) (TLA2) and (STL4) to instantiate the refinement mapping and verify a transformation is captured by $\Rightarrow_T$:

$$\frac{\langle\theta',\eta'\rangle \Rightarrow \langle\theta,\overline{\eta}\rangle \qquad [run_{bcs'}]_{\langle\theta',\eta'\rangle} \Rightarrow [run_{\overline{bcs}}]_{\langle\theta,\overline{\eta}\rangle}}{\langle\theta',\eta',bcs'\rangle \Rightarrow_T \langle\theta,\eta,bcs\rangle}.$$

An important feature, which underpins the calculus, is the transitivity of $\Rightarrow_T$:

Theorem 8.1. $\langle\theta,\eta,bcs\rangle \Rightarrow_T \langle\theta',\eta',bcs'\rangle$ *and* $\langle\theta',\eta',bcs'\rangle \Rightarrow_T \langle\theta'',\eta'',bcs''\rangle$ *implies* $\langle\theta,\eta,bcs\rangle \Rightarrow_T \langle\theta'',\eta'',bcs''\rangle$.

Proof outline. The proof reduces to transitivity of $\Rightarrow$ which is trivial. ∴

8.3 Rules & strategies

Gen_Rules(rs)**:** Returns a *generalisation* of *rs*. In patterns variables are replaced by '_' while the rest is unchanged. In expression everything but '$*$' is replaced by '_', and all function calls are removed.
get_box(B, bcs)**:** Returns box configuration with box id B from list *bcs*.
HeapLocs_Copy$\big([l_1,\cdots l_n],H_1,H_2\big)$**:** Returns a tuple $\langle[l'_1,\cdots l'_n],H'_2\rangle$ holding a copy of $[l_1,\cdots l_n]$ of H_1 into H_2 and the updated H_2.
is_Blocked(B)**:** Holds if box B cannot be executed.
len(L)**:** Returns the length of list L.
mutually_exclusive(rs)**:** Holds if the patterns of rule set *rs* are mutually exclusive.
project$\big([(p_1 \to e_1),\cdots(p_n \to e_n)],[(p'_1 \to e'_1),\cdots(p'_m \to e'_m)]\big)$**:** Pairwise combines each pattern p_i and p'_j with e_i and e'_j where $i \in 1..n$ and $j \in 1..m$.
$L_1@L_2$**:** Concat list L_1 in front of list L_2.
Time_Dependency(B, bcs)**:** Predicate that holds iff changing the number of steps in B cannot alter the overall behaviour when running program defined by box configuration *bcs*.

Figure 8.2: Box calculus auxiliary functions

The general categories of transformation rules in the box calculus will be familiar from many comparable calculi. There are rules to: introduce/eliminate identity boxes; introduce/eliminate nesting boxes; introduce/eliminate wires; combine/separate boxes horizontally and vertically; expand/contract match patterns and results; and reorder patterns and results. Special to Hume are rules for moving activity between result expressions within boxes and coordination between boxes. Indeed, in Hume, coordination and expression level transformation are tightly coupled, and there are necessarily strong links between the apparently distinct categories above.

A full "formal" definition of all the rules will not be given, since the focus is on the approach rather than all the underlying details. Moreover, a full derivation of all rules would require more space than just a chapter. This discussion is therefore limited to just a few rule and strategy derivations and a sketch of their correctness proofs. The remaining rules and strategies are summarised below. Details, such as rule pre conditions, have there been omitted. The rule derivations also requires some auxiliary functions, which do not have any side effects on the program configuration. These are shown in Figure 8.2.

Standard logical terminology is used in the rules: a rule postfixed by 'I' is a rule that "introduces something", and its dual, the elimination rule, is postfixed by 'E'. The rule derivations are also augmented by a graphical representation which only shows the direct coordination impact of the rules.

In Chapters 6 and 7 it was shown that box nesting greatly mitigates the impact of a box transformation, since timing contraints are localised and may be considered independently of the rest of the program. The elements where such a transformation may have an impact is called the box *context*, and include the siblings, parents, and the box itself, together with the wires connecting them.

8.3.1 Summary of rules & strategies

CaseE(B, i) **:** Moves case expression in match i of box B into B's rule set

CaseI(B, i, j) **:** Replaces match i to j in box B by a case-expression.

DupE(B, x, y) **:** Removes wire connected to x of B which is equal to wire connected to y of box B.

DupI(A, x, x', B, y, y') **:** Duplicates wire connecting x of box A and y of B, with wire names x' (of A) and y' (of B) respectively.

DupBoxE(A, B) **:** Eliminates box A which is a duplicate of box B.

DupBoxI(B, N) **:** Duplicates box B, which is given name N.

HCompE$(B, [i_1, \cdots, i_n], [o_1, \cdots, o_m], X, Y)$ **:** Horizontally de-composes box B into boxes X and Y, where X has inputs $[i_1, \cdots, i_n]$ and outputs $[o_1, \cdots, o_m]$. Y will have the inputs/output of B not in $[i_1, \cdots, i_n]/[o_1, \cdots, o_m]$.

HCompI(A, B, N)**:** Horizontally composes box A and box B into N.

HieE(B)**:** Replaces B with it's (only) child box.

HieI(B, N)**:** Replaces box B by N which only holds B.

IdE(B)**:** Eliminates identity box B.

IdI(B, v, N)**:** Introduces an identity box N to wire connected to v of box B.

MatchE(B, n)**:** Eliminates match n of box B.

MatchI(B) **:** Introduces an empty match to box B.

MatchVarE(B, v) **:** Replaces variable v in box B by constants (unfolds the finite type).

MatchVarI(B, i, o)**:** Replaces constants in inputs i and output o by a variable.

Rename(A, N)**:** Renames box A to N.

Replace$\big([A_1, \cdots, A_n], [B_1, \cdots, B_m]\big)$ **:** Replaces boxes $A_1, \cdots, A_n$ by $B_1, \cdots, B_m$.

ReplaceExpr(A, n, e) **:** The expression of match n of box A is replaced by e.

ThreadE(B, x, y) **:** Removes threading of input x and output y of box B.

ThreadI(A, x, B) **:** Threads wire connected to x of box A through box B.

Unfold(B, n, f) **:** Unfolds function f in match n of box B.

ValueE(B, x) **:** Replaces wire w of box B by '$*$'.

ValueI1(A, x, B, y, v) **:** Introduces value v to expressions of wire x of box A and pattern of wire y of box B.

ValueI2(A, x, v, B, y, w) **:** Copies all expression of wire v to wire x of box A and all of wire w to wire y of box B.

VCompE$(B, N, [o_1, \cdots o_n], M, [i_1, \cdots i_n])$ **:** Vertically de-composes box B into two sequentially composed boxes N and M, where N has B's inputs and $[o_1, \cdots o_n]$ as outputs, and M has $[i_1, \cdots i_n]$ as inputs and B's outputs.

VCompI(A, B, N) **:** Vertically composes box A and box B into a new box with name N.

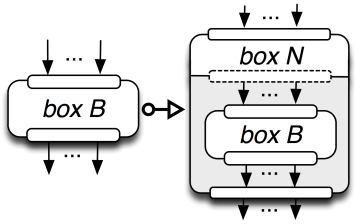
VRename(A, x, N)**:** Renames wire x of box A to N.

WireE(B, x) **:** Eliminates (empty) wire connected to wire x of box B.

WireI(A, x, B, y) **:** Introduces an empty wire holding only '$*$' patterns/ expressions from (new) wire x of box A to (new) wire y of box B.

8.3.2 Derivation of HieI

The first rule, **HieI**, nests one box with name B inside another box A with name N. This rule introduces a *bounded context* for B, only consisting of N and B. By applying this rule, the top level timing dependencies can be ignored when transforming B, and many (temporal) preconditions of rules require a bounded context. The rule copies input and output wires to the internal heap, by using the **HeapLocs_Copy** function. These are the new wires of the newly created nested box B', and the internal wires of the nesting box A. Further, A consists of one nested box B' and *generalises* B's rule set into the more restricted hierarchical form, by **Gen_Rules**:



$$\frac{\begin{array}{c}\langle B, iws, ows, rs, iw, nci, ow, ibcs\rangle = \textbf{get\_box}(B, bcs) \\ \langle niw, \eta''\rangle = \textbf{HeapLocs\_Copy}(iws, \theta, \eta) \\ \langle now, \eta'\rangle = \textbf{HeapLocs\_Copy}(ows, \theta, \eta'') \\ B' = \langle B, niw, now, rs, iw, nci, ow, ibcs\rangle \qquad irs = \textbf{Gen\_Rules}(rs) \\ A = \big\langle N, iws, ows, irs, niw, [], now, [B']\big\rangle \\ \langle \theta, \eta, bcs\rangle \vdash \textbf{Replace}\big([A], [B]\big) \Downarrow \langle \theta, \eta', bcs'\rangle\end{array}}{\langle \theta, \eta, bcs\rangle \vdash \textbf{HieI}(B, N) \Downarrow \langle \theta, \eta', bcs'\rangle}$$

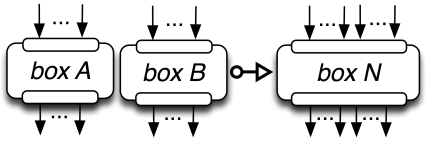
Theorem 8.2.

$$\begin{array}{ll} \textit{If} & \langle \theta, \eta, bcs\rangle \vdash \textbf{HieI}(A, N) \Downarrow \langle \theta', \eta', bcs'\rangle \\ \textit{then} & \langle \theta', \eta', bcs'\rangle \Rightarrow_T \langle \theta, \eta, bcs\rangle \end{array}$$

Proof outline. $\overline{\eta}$ is only extended and θ is not changed, thus $\langle \theta', \overline{\eta}'\rangle \Rightarrow \langle \theta, \overline{\eta}\rangle$ holds. In *bcs*, B is replaced by A. Since A's rule set generalises B's the matching will be the same. Further, since A only contains B, the computation and termination will be the same, and therefore also the result. Therefore $run_{\overline{bcs'}} \Rightarrow run_{\overline{bcs}}$ holds. $\therefore$

8.3.3 Derivation of HCompI

In the derivation of **HCompI**, two non-nested boxes, A and B, are horizontally composed into a new box called N. However, A and B must always have the same *Blocked* status, since N will be *Blocked* if either of them is. If one, but not the other, is *Blocked* the behaviour of the composed box N will not capture the sum of A and B. The inputs and outputs of A prefix B's inputs and outputs. For all matches, the patterns and expressions of A and B are pairwise composed by **project**. This projection might introduce non-determinacy, so the patterns must be mutually exclusive. Finally, A might execute while B fails to pattern match the inputs, and vice verse. This is captured by postfixing the composed rule set below with a rule set where A's rule set is composed with only '$*$'s, and the same for B. The box N', capturing all the above, replaces A and B:



$$\frac{\begin{array}{c}\langle A, iws_A, ows_A, rs_A, [], [], [], []\rangle = \textbf{get\_box}(A, bcs)\\ \langle B, iws_B, ows_B, rs_B, [], [], [], []\rangle = \textbf{get\_box}(B, bcs)\\ \Box\big(\textbf{is\_Blocked}(A) \equiv \textbf{is\_Blocked}(B)\big)\\ \textbf{mutually\_exclusive}(rs_A) \qquad \textbf{mutually\_exclusive}(rs_B)\\ iws = iws_A @ iws_B \qquad ows = ows_A @ ows_B\\ n_A = \textbf{len}(iws_A) \quad m_A = \textbf{len}(ows_A)\\ n_B = \textbf{len}(iws_B) \quad m_B = \textbf{len}(ows_B)\\ *_A = [\underbrace{\langle *, \cdots, * \rangle}_{n_A} \rightarrow \underbrace{\langle *, \cdots, * \rangle}_{m_A}] \quad *_B = [\underbrace{\langle *, \cdots, * \rangle}_{n_B} \rightarrow \underbrace{\langle *, \cdots, * \rangle}_{m_B}]\\ rs = \textbf{project}(rs_A, rs_B) \ @ \ \textbf{project}(rs_A, *_B) \ @ \ \textbf{project}(*_A, rs_B)\\ N' = \langle N, iws, ows, rs, [], [], [], []\rangle\\ \langle \theta, \eta, bcs\rangle \vdash \textbf{Replace}([N'], [A, B]) \Downarrow \langle \theta, \eta, bcs'\rangle\end{array}}{\langle \theta, \eta, bcs\rangle \vdash \textbf{HCompI}(A, B, N) \Downarrow \langle \theta, \eta, bcs'\rangle}$$

The unification with the empty list ensures that the boxes are not nested with **get_box**. The mutual exclusiveness test is straightforward, and the establishment of the *Blocked* state requires a temporal invariance proof. This has therefore been prefixed with $\Box$.

Theorem 8.3.

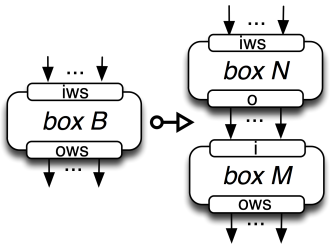
$$\begin{array}{ll} \textit{If} & \langle \theta, \eta, bcs\rangle \vdash \textbf{HCompI}(A, B, N) \Downarrow \langle \theta, \eta, bcs'\rangle \\ \textit{then} & \langle \theta', \eta', bcs'\rangle \Rightarrow_T \langle \theta, \eta, bcs\rangle \end{array}$$

Proof outline. There is no nesting, hence $\overline{\eta} = \eta$. Further, it is obvious that $\theta' = \theta$ and $\eta' = \eta$, thus $\langle \theta', \overline{\eta'}\rangle \Rightarrow \langle \theta, \overline{\eta}\rangle$. The proof of $run_{\overline{bcs'}} \Rightarrow run_{\overline{bcs}}$ is by case-analysis on the box state of A and B: Since $\Box\big(\textbf{is\_Blocked}(A) \equiv \textbf{is\_Blocked}(B)\big)$, A and B are always *Blocked* at the same time. Hence, if one is *Blocked* then so is the other, and since N will be *Blocked* if either of them are, so is N. If both A and B succeed then, since all possible matches are composed, so will N. Since the patterns are mutually exclusive only one pattern can succeed, and the result is obviously the same. If both boxes fail to execute, then so will N since it only composes A and B. Finally, the case where only one box succeeds is captured by the case where each match is composed with only '$*$'s. Thus the goal holds. $\therefore$

8.3.4 Derivation of VCompE

The third, and final, rule **VCompE**, vertically decomposes a box: it assumes a box B with one match, where the expression has the form of a function composition of functions f and g on the input. The box is vertically split between f and g such

as the result of f in new box N, is the input on the wires to new box M, which has the expression g. However, this split introduces an extra step. The predicate **Time_Dependency**(B, bcs) holds if changing the number of steps to compute B may change behaviour of the full program, defined by *bcs*. Hence, the negation of **Time_Dependency**, ensures that the extra steps do not change the behaviour:



$$\frac{\begin{array}{c}\neg\textbf{Time\_Dependency}(B, bcs) \\ \langle B, iws_B, ows_B, rs_B, [], [], [], [] \rangle = \textbf{get\_box}(B, bcs) \\ rs_B = [\langle v_1, \cdots, v_m \rangle \rightarrow g(f\langle v_1, \cdots, v_m \rangle)] \\ rs_N = [\langle v_1, \cdots, v_m \rangle \rightarrow f\langle v_1, \cdots, v_m \rangle] \\ rs_M = [\langle i_1, \cdots, i_n \rangle \rightarrow g\langle i_1, \cdots, i_n \rangle)] \\ N = \langle N, iws, [o_1, \cdots, o_n], rs_N, [], [], [], [] \rangle \\ M = \langle M, [i_1, \cdots, i_n], ows, rs_M, [], [], [], [] \rangle \\ \langle \theta, \eta, bcs \rangle \vdash \textbf{Replace}([B], [N, M]) \Downarrow \langle \theta, \eta, bcs' \rangle\end{array}}{\langle \theta, \eta, bcs \rangle \vdash \textbf{VCompE}(B, N, [o_1, \cdots, o_n], M, [i_1, \cdots, i_n]) \Downarrow \langle \theta, \eta, bcs' \rangle}$$

Theorem 8.4.

$$\begin{array}{ll} \textit{If} & \langle \theta, \eta, bcs \rangle \vdash \textbf{VCompE}(B, N, [o_1, \cdots, o_n], M, [i_1, \cdots, i_n]) \Downarrow \langle \theta, \eta, bcs' \rangle \\ \textit{then} & \langle \theta', \eta', bcs' \rangle \Rightarrow_T \langle \theta, \eta, bcs \rangle \end{array}$$

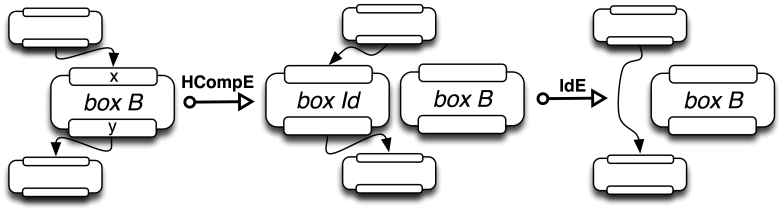
Proof outline. The rule represents function de-composition lifted up to the coordination layer. The wiring ensures sequential execution. Thus showing that computing f then g in one cycle, will produce the same result as f in one cycle and then g in the next cycle, is trivial. The de-composition introduces an extra cycle which may change the overall behaviour of the context. However, the predicate $\neg$**Time_Dependency**(B, bcs) ensures that this is not the case. ∴

8.3.5 Strategies

The rules will often be too low-level to work with. Instead a user will work with higher-level *strategies*, which are derived from rules and other strategies. An example of a strategy, although still rather low-level, is the elimination of *threading*. A wire is threaded through a box if there is a one-to-one correspondence between a pattern x and an expression y in all matches. x cannot be used in other expressions (except y).

Further, x and y must form an identity box. When eliminated, the threaded value will arrive earlier at the destination. This must not have any effect on the context. Finally, a *Blocked* state on B will prevent the threaded value leaving B, which is not the case when the thread is eliminated. This must again not have any impact on the context. Since the rule is derived from other rules these precondition can be ignored as they are implicitly captured by the preconditions of the rules in the derivation.

For example threading elimination, **ThreadE**, is derived as follows: x and y are horizontally de-composed into a new box *Id* by **HCompE**; *Id* is then an identity box eliminated by **IdE**:



$$\frac{\langle\theta,\eta,bcs\rangle \vdash \mathbf{HCompE}(B,[x],[y],Id,B) \Downarrow \langle\theta_1,\eta_1,bcs_1\rangle \quad \langle\theta_1,\eta_1,bcs_1\rangle \vdash \mathbf{IdE}(Id) \Downarrow \langle\theta',\eta',bcs'\rangle}{\langle\theta,\eta,bcs\rangle \vdash \mathbf{ThreadE}(B,x,y) \Downarrow \langle\theta',\eta',bcs'\rangle}$$

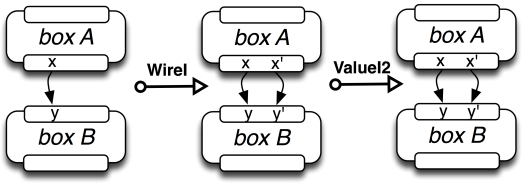
The correctness proof for strategies are trivial since they only rely on Theorem 8.1:

Theorem 8.5.

If $\quad \langle\theta,\eta,bcs\rangle \vdash \mathbf{ThreadE}(B,x,y) \Downarrow \langle\theta',\eta',bcs'\rangle$
then $\quad \langle\theta',\eta',bcs'\rangle \Rightarrow_T \langle\theta,\eta,bcs\rangle$

Proof outline. Since the two given rules are applied sequentially, the proof reduces to the transitivity of $\Rightarrow_T$. This is proved by Theorem 8.1. $\therefore$

The second strategy derived, **DupI**, shows how to duplicate a wire: The wire connecting variable x of box A with variable y of box B is copied to a new wire between variable x' and y' of boxes A and B, respectively. The same initial value is given, and the expression of x in A are copied to x', while the matches of y in B are copied to y':



$$\frac{\langle\theta,\eta,bcs\rangle \vdash \mathbf{WireI}(A,x',B,y') \Downarrow \langle\theta_1,\eta_1,bcs_1\rangle \quad \langle\theta_1,\eta_1,bcs_1\rangle \vdash \mathbf{ValueI2}(A,x',x,B,y',y) \Downarrow \langle\theta',\eta',bcs'\rangle}{\langle\theta,\eta,bcs\rangle \vdash \mathbf{DupI}(A,x,x',B,y,y') \Downarrow \langle\theta',\eta',bcs'\rangle}$$

Theorem 8.6.

$$\begin{array}{ll} \textit{If} & \langle \theta, \eta, bcs \rangle \vdash \mathbf{DupI}(A, x, x', B, y, y') \Downarrow \langle \theta', \eta', bcs' \rangle \\ \textit{then} & \langle \theta', \eta', bcs' \rangle \Rightarrow_T \langle \theta, \eta, bcs \rangle \end{array}$$

Proof outline. The proof is the same as in Theorem 8.5. ∴

8.4 Examples

Application of the calculus is illustrated by two examples. The first example shows the decomposition of a half adder box into a binary tree of three elementary logic gate boxes. The second example shows the decomposition of a full adder box into two half adders and an elementary logic gate.

8.4.1 Example 1: half adders

Figure 8.3.a shows the source box `half1` of the transformation, while Figure 8.3.b and Figure 8.3.c show the resulting `half2` box in diagrammatical and source code form. The overall impact of the translation is from a truth table representation into a nesting box containing a multi-box AND/XOR configuration. The calculus is applied stepwise in a forward style, starting with `half1`. A dot '.' notation is used to refer to nested boxes, starting from the first level. If a rule has more than one parameter, it is sufficient to give the full path to one of the boxes, since a rule is only applied to one context at a time. To ease reading, the program configuration triple is omitted. A graphical representation of each step is shown in Figure 8.4:

1. Rule **HieI**(`half1`, `half2`) replaces box `half1` with a box `half2` that nests `half1`.

2. There are no $*$'s in the context nested by `half2`. This implies that there are no dependencies, and identity boxes for both input wires of `half1` can be introduced: **IdI**(`half2.half1`, `x`, `Id`) followed by **IdI**(`half2.half1`, `y`, `Id'`). The input/output variables of the identity boxes are `v/v'` by default. These are renamed to `x/x1` and `y/y1` respectively: **VRename**(`half2.Id`, `v`, `x`), **VRename**(`half2.Id`, `v'`, `x1`), **VRename**(`half2.Id'`, `v`, `y`) and **VRename**(`half2.Id'`, `v'`, `y1`).

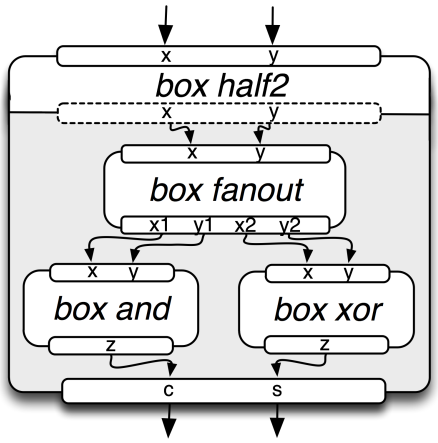
3. The two identity boxes are then horizontally composed into one box called `fanout`: **HCompI**(`half2.Id`, `Id'`, `fanout`):

```
box fanout
  in (x,y::Bit)  out (x1,y1::Bit)
match
```

```
-- Only 0 and 1
type Bit = int 1;

box half1
  in (x,y::Bit)
  out (s,c::Bit)
match
 (0,0) -> (0,0) |
 (0,1) -> (1,0) |
 (1,0) -> (1,0) |
 (1,1) -> (0,1);
```

a) `half1`*: truth table.*



b) `half2`: `xor` *and* `and` *gates box diagram.*

```
box half2
   in (x,y::Bit)  out (s,c::Bit)
match
  (_,_) -> (_,_)
boxes
   box fanout
      in (x,y::Bit)
      out (x1,y1,x2,y2::Bit)
   match
       (x,y) -> (x,y,x,y);
   wire fanout (half1.x,half2.y)
               (xor.x,xor.y,and.x,and.y);
   box xor
       in (x,y::Bit)  out (z::Bit)
   match
      (0,0) -> 0 |
      (0,1) -> 1 |
      (1,0) -> 1 |
      (1,1) -> 0;
   wire xor(fanout.x1,fanout.y1)
             (half1.s);
   box and
      in (x,y::Bit)  out (z::Bit)
   match
     (0,0) -> 0 |
     (0,1) -> 0 |
     (1,0) -> 0 |
     (1,1) -> 1;
   wire and (fanout.x2,fanout.y2)
            (half1.c);
end;
```

c) `half2`: *source code.*

Figure 8.3: Half adders in Hierarchical Hume.

```
(x,y)->(x,y) | (x,*)->(x,*) | (*,y)->(*,y) ;
```

A simple invariant of the internal behaviour of `half2` shows that it will never be the case that only one of `fanout`'s inputs is empty. The last two matches of `fanout` will therefore never succeed. This is the only precondition in the match elimination rule which can therefore be applied: **MatchE**(`half2.fanout`, 3) and **MatchE**(`half2.fanout`, 2).

4. The two wires connecting `fanout` and `half1` are duplicated and named `x2` and `y2`: **DupI**(`half2.fanout`, `x1`, `x2`, `half1`, `x`, `x2`) followed by **DupI**(`half2.fanout`, `y1`, `y2`, `half1`, `y`, `y2`).

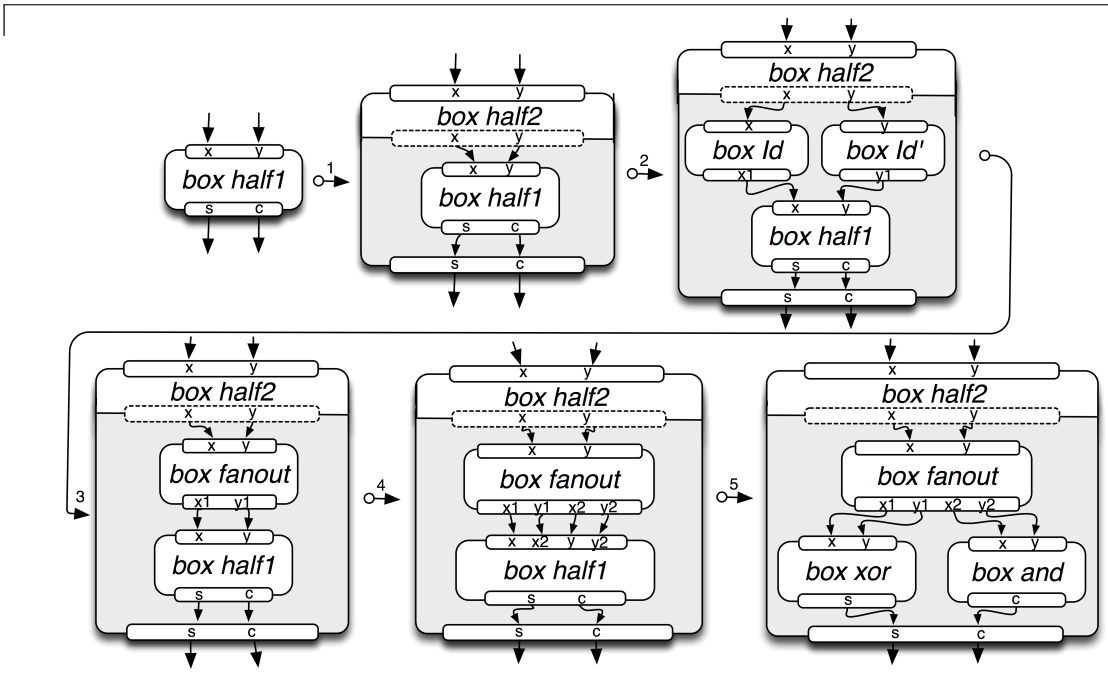


Figure 8.4: Transformation of half adder.

5. In `half1` there are two sets of identical inputs: $\{\texttt{x}, \texttt{y}\}$ and $\{\texttt{x2}, \texttt{y2}\}$. The output `s` can be seen to depend only on the first set, while `c` only depends on the second. The box can thus be de-composed. The first of these boxes is exactly the same as the `xor` while the second is the same as the `and` box of Figure 8.3(b/c): **HCompE**(`half2.half1`, [`x`,`y`], [`s`], `xor`, `and`). Finally, the inputs of the `and` box are renamed: **VRename**(`half2.and`, `x2`, `x`) and **VRename**(`half2.and`, `y2`, `y`).

8.4.2 Example 2: full adders

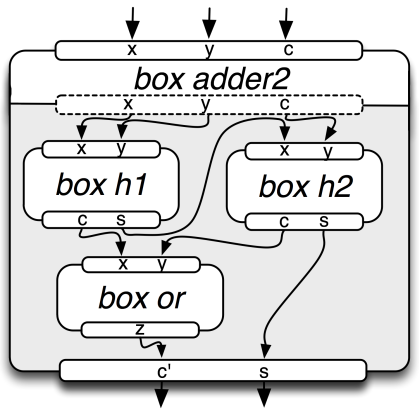
The second example is more complex: a full adder `adder1`, represented as a truth table (Figure 8.5a) is transformed into a representation `adder2` using two half adders and an OR gate (Figure 8.5b/c). As above, the transformation is step-by-step and each step is graphically illustrated in Figure 8.6:

1. First, all matches of `adder1` are moved inside a case expression. Since the patterns are total with respect to the `Bit` type this is allowed: **CaseI**(`adder1`, `1`, `8`):

```
box adder1
  in (x,y,c::Bit) out (s,c'::Bit)
match
  (a,b,c) -> case (a,b,c) of  ...;
```

```
box adder1
in (x,y,c::Bit)
out (s,c'::Bit)
match
 (0,0,0) -> (0,0) |
 (0,1,0) -> (1,0) |
 (1,0,0) -> (1,0) |
 (1,1,0) -> (0,1) |
 (0,0,1) -> (1,0) |
 (0,1,1) -> (0,1) |
 (1,0,1) -> (0,1) |
 (1,1,1) -> (1,1) ;
```

a) `adder1`: *truth table.*



b) `adder2`: `half` *adders and* `or` *gate.*

```
box adder2
   in (x,y,c::Bit)  out (s,c'::Bit)
match
  (_,_,_) -> (_,_)
boxes
   box h1
      in (x,y::Bit) out (s,c::Bit)
   match
     (0,0) -> (0,0) |
     (0,1) -> (1,0) |
     (1,0) -> (1,0) |
     (1,1) -> (0,1);
  wire h1(adder2.x,adder2.y)(h2.x,or.x);

  box h2
      in (x,y::Bit) out (s,c::Bit)
  match ... -- same as h1
  wire h2(h1.c,adder2.c)(adder2.s,or.y);

  box or
     in (x,y::Bit)  out (z::Bit)
  match
    (0,0) -> 0 |
    (0,1) -> 1 |
    (1,0) -> 1 |
    (1,1) -> 1;
  wire or(h1.c,h2.c)(adder2.c);
end;
```

c) `adder2`: *source code.*

Figure 8.5: Full adders in Hierarchical Hume.

The case expression is then replaced by the function composition `g · f(a,b, c)`: **ReplaceExpr**(`adder1`, 1, `g · f(a,b,c)`) where `f` and `g` are shown in Figure 8.7. The next step vertically de-composes this box, where f is the expression of the first box and g the expression of the second box. However, this will introduce an extra step, and since the context is unknown, the `adder1` box is first nested: **HieI**(`adder1`, `adder2`). The boxes can then safely be de-composed: **VCompE**(`adder2.adder1`, `h1h2`, [`s,x',c'`], `or`, [`z,x,y`]).

2. The newly created `or` box has one match with the expression `g`, where `g` consists of a (total) case expression. `g` is unfolded and the case-expression is moved into the match: **Unfold**(`adder2.or`, `g`) followed by **CaseE**(`adder2.or`, 1). The result is illustrated on the left side below. The first pattern and expression are identical (and total), and may thus be replaced by a variable: **Match-**

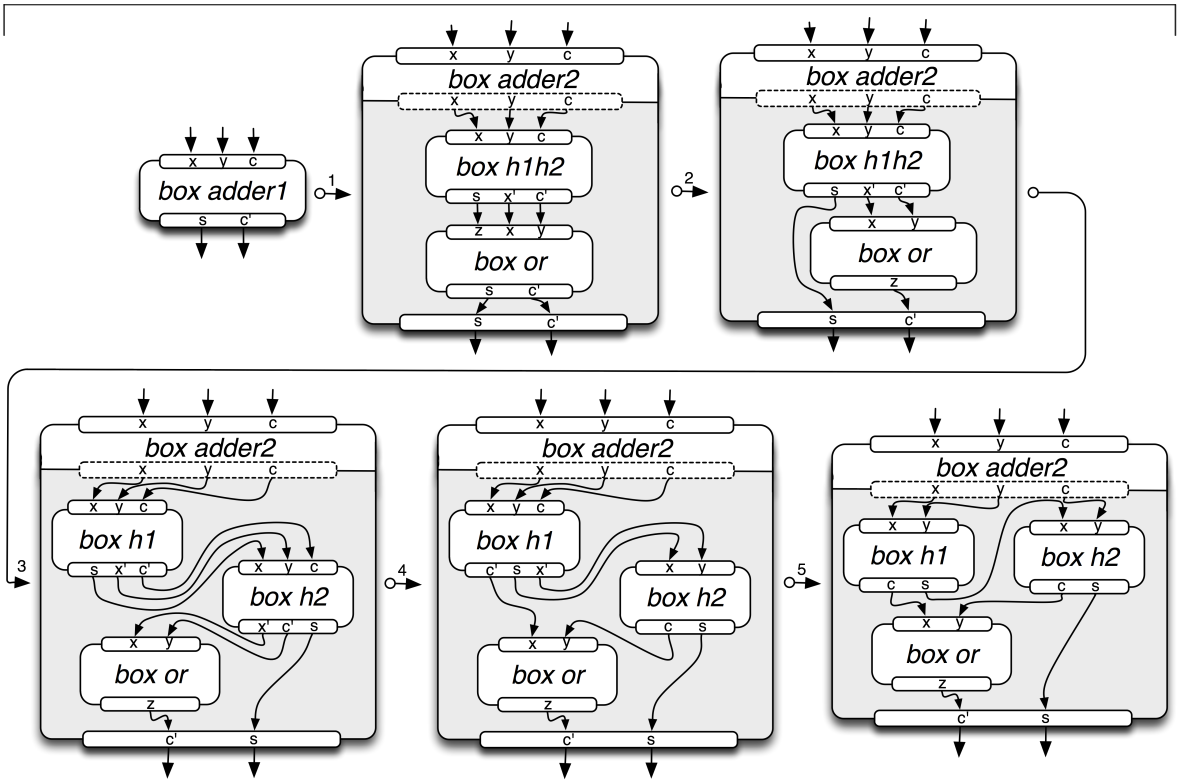


Figure 8.6: Transformation of full adder.

VarI(`adder2.or`, `x`, `s`). A variable is now threaded and can be eliminated (since there are no $*$'s in the context): **ThreadE**(`adder2.or`, `x`, `s`). The result is illustrated on the right side:

```
box or
 in(z,x,y::Bit) out(s,c'::Bit)
match
   (0,0,0) -> (0,0) |
   (1,0,0) -> (1,0) |
   (0,0,1) -> (0,1) |
   (1,0,1) -> (1,1) ...;
```

```
box or
 in(x,y::Bit) out(c'::Bit)
match
   (0,0) -> 0 |
   (0,0) -> 0 |
   (0,1) -> 1 |
   (0,1) -> 1 ...;
```

Matches $2, 4, 6$ and 8 are now duplicates of their previous matches, and can therefore be removed: **MatchE**(`adder2.or`, 8),**MatchE**(`adder2.or`, 6), **MatchE**(`adder2.or`, 4) and **MatchE**(`adder2.or`, 2). Finally, the output wire is renamed to `z`: **VRename**(`adder2.or`, `c'`, `z`). The `or` box is now the same as in Figure 8.5c.

3. Box `h1h2` consists of one match with expression `f`. This function can be replaced by the function composition `gg·ff(a,b,c)` where `ff` and `gg` are shown in Figure 8.7: **ReplaceExpr**(`adder2.h1h2`, 1, `gg·ff(a,b,c)`). Since the context does not

```
f(a,b,c) = case (a,b,c) of
           (0,0,0) -> (0,0,0) |
           (0,0,1) -> (1,0,0) |
           (0,1,0) -> (1,0,0) |
           (0,1,1) -> (0,0,1) ...;

g(a,b,c) = case (a,b,c) of
             (0,0,0) -> (0,0) |
             (1,0,0) -> (1,0) |
             (0,0,1) -> (0,1) |
             (1,0,1) -> (1,1) ...;

ff(a,b,c) = case (a,b,c) of
            (0,0,0) -> (0,0,0) |
            (0,0,1) -> (1,0,0) |
            (0,1,0) -> (0,1,0) |
            (0,1,1) -> (1,1,0) ...;

gg(a,b,c) = case (a,b,c) of
              (0,0,0) -> (0,0,0) |
              (0,0,1) -> (0,1,0) |
              (0,1,0) -> (1,0,0) |
              (0,1,1) -> (1,1,0) ...;
```

Figure 8.7: Auxiliary functions required in the full adder transformation.

contain any $*$'s vertical function de-composition can be applied: **VCompE**(adder2.h1h2, h1, [s,x',c'], h2, [x,y,c]).

4. gg of box h2 is unfolded and the (total) case-expression is moved into the body: **Unfold**(adder2.h2, gg) and **CaseE**(adder2.h2, 1) as illustrated on the left side below. The last pattern and second expression of all matches can be replaced by a variable, which creates a threading that can be eliminated: **MatchVarI**(adder2.h2, c, s) and **ThreadE**(adder2.h2, c, s), as illustrated on the right side:

```
box h2
 in(x,y,c::Bit) out(s,x',c'::Bit)
match
   (0,0,0) -> (0,0,0) |
   (0,0,1) -> (0,1,0) |
   (0,1,0) -> (1,0,0) |
   (0,1,1) -> (1,1,0) ...;
```

```
box h2
 in(x,y::Bit) out(s,c'::Bit)
match
   (0,0) -> (0,0) |
   (0,0) -> (0,0) |
   (0,1) -> (1,0) |
   (0,1) -> (1,0) ...;
```

Matches 2, 4, 6 and 8 are now duplicates of previous matches and therefore removed: **MatchE**(adder2.h2, 8), **MatchE**(adder2.h2, 6), **MatchE** (adder2.h2, 4) and **MatchE**(adder2.h2, 2). By renaming the last output to c' a correct implementation of a half adder is created: **VRename** (adder2.h2,c',c).

5. The transformation of h1 follows the same pattern as h2 (and or): First the case expression is removed, followed by a variable introduction and threading elimination: **Unfold**(adder2.h1, ff), **CaseE**(adder2.h1, 1), **MatchVarI** (adder2.h1, c, x') and **ThreadE**(adder2.h2, c, x'). Then the duplicate matches are removed, which creates a correct implementation of a *half*-adder: **MatchE** (adder2.h1, 8), **MatchE**(adder2.h1, 6), **MatchE**(adder2.h1, 4) and **MatchE**

(`adder2.h1`, 2). By renaming `c'` to `c` the transformation is concluded: **VRename**(`adder2.h1`,`c'`,`c`). An even lower representation can be achieved by applying the half-adder transformation to `h1` and `h2` as explained above.

8.5 Related work

A full Hume *transformation* combines rules and strategies, and is thus a strategy. As discussed in Chapter 6, and following Visser's classification [192], it can be seen both as a *program migration* and a *program refinement*.

A single rule application which is not merely program rephrasing of the expression layer will have an impact on both layers. This strong interplay between the two aspects of a Hume program, is distinctive compared with synthesis techniques, like Bird-Meertens Formalism [26] and calculational programming [99]. Moreover, a generic rule has no particular direction with respect to the two layers, and should thus be classified as a *program refactoring*. Many strategies, in particular the lower-level, should also be seen as program refactorings.

Following work with other refinement based formalisms like B [8], Event-B [9], and probably also the ZRC [40] refinement calculus for Z, applying many small refinements, instead of one large refinement greatly helps automating the proof in a theorem prover. This is for example illustrated in [39] for Event-B. The box calculus draws upon this, since a single rule has a small impact on the overall program, and a strategy is a sequence of such small rule steps. High-level strategies are comparable to newly introduced *refinement patterns* [100], which are used to capture common refinements. However, a strategy is probably more detailed due to the context being a programming language rather than a more abstract specification.

Just as Hume integrates a finite state coordination language with a functional transition control language, the work presented here draws on the twin traditions of process network and functional program transformation. The coordination aspects of the rules have many similarities with those found in the box calculus for Petri nets [54] as well as process calculi [18]. The control aspects resemble classic functional programming techniques including curry/uncurry, fold/unfold [36] and functional refactoring [134]. Previously, horizontal box integration has been explored in establishing informally that FSM-Hume actually is finite state [146].

8.6 Summary & discussion

A calculus supporting the systematic transformation of program components through the introduction, modification, elimination, composition and separation of boxes and wires, has been outlined and applied to examples. A major strength and novelty of the calculus is that it combines changes to control aspects within boxes with those to coordination aspects between boxes.

The next chapter discusses three separate topics, which are relevant to the main topic of this thesis.

Chapter 9

Relevant explorations

9.1 Introduction

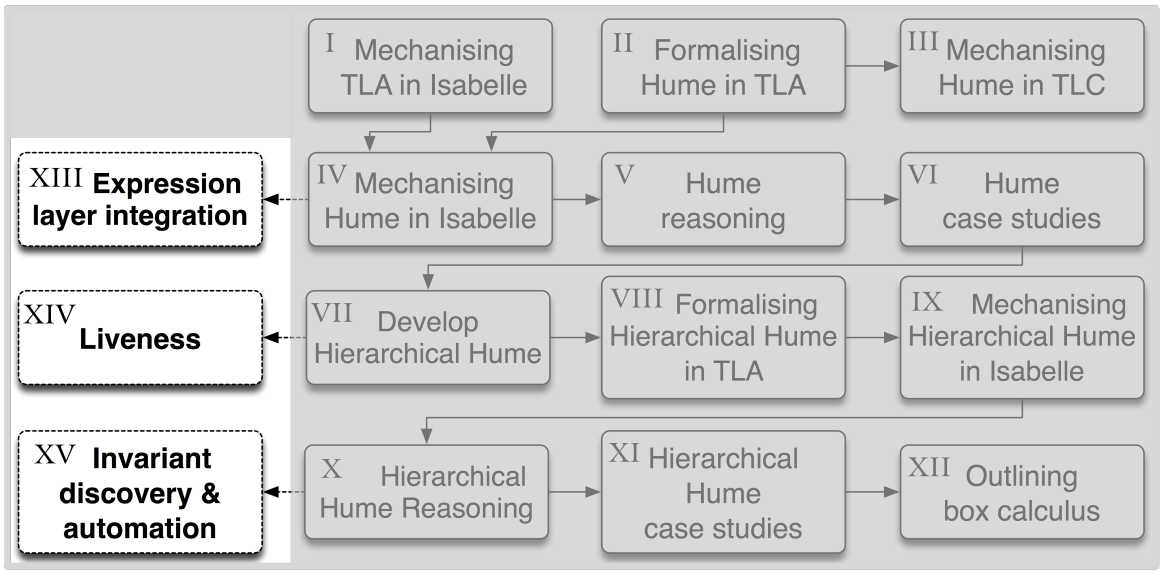


Figure 9.1: Thesis roadmap: Chapter 9

This chapter contains three separate explorations which are relevant to the topic of the thesis, but not on the critical path. In the thesis roadmap these parts represents the dotted boxes, as highlighted in Figure 9.1.

One of the motivation for the use of theorem proving, and particularly Isabelle/HOL, was that this work is seen as a first step towards a Hume verification environment, and should be integrated with the expression layer mechanisation conducted in parallel. In Section 9.2, an initial experiment of this integration is shown.

A key feature of TLA, is that it can capture liveness and safety properties within the same uniform logic. This thesis focuses on the safety aspect. Section 9.3 discusses

how this can be extended to liveness.

The tactics developed in this thesis has a high degree of automation if the conjecture is sufficiently strong. However, in most cases the conjecture had to be strengthened by several auxiliary lemmas, in a "weakest precondition" fashion. This had to be achieved manually. Section 9.4 outlines an approach based on a proof plan called rippling [33] to automatically discover (D3) type invariants.

9.2 Integrating the expression layer[1]

This section discusses the integration with the VDM (Vienna Development Method) [108] based mechanisation of the expression layer [135]. For the purpose of the work here, the VDM logic provides pre-condition and post-condition style properties (similar to Hoare logic) of the expression layer. These are then used in the proofs within the TLA embedded coordination layer. The most relevant background is first given, before the integration is explained and shown for an example. Note that Isabelle 2005 is used, requiring an adaption of Isabelle/Hume, thus disabling the generated tactics.

9.2.1 Background: a VDM-style logic for the expression layer

The allowed types are represented by an inductive data type HoVal. Here, primitive types like int and bool are embedded shallowly. The empty type $\bot$ and reference types to the heap are also supported. The heap is represented by mechanising a finite map with look-up and update functions. It is represented by the type acronym Heap, with actual type nat $\Rightarrow$ HoVal.

The operational semantics for an expression e are given a deep inductive embedding, and are written E, h $\vdash$ e $\Downarrow$ (v,hh,p): given environment E and heap h, the expression e returns a value (HoVal) v, a new heap hh and the required resources p. On top of this, a higher-level VDM-style logic is derived where a judgement has the form G $\rhd$ e : A: G is the context; while A is the assertion. The judgement denotes that A holds for e in context G. The assertion is mechanised shallowly, and has the type λ E h hh v p. P(E,h,hh,v,p). The shallow representation is the key for a direct integration. The integration assumes an empty context {}, thus G is not discussed further. Firstly, *valid in context* is defined as G $\vDash$ e : A $\equiv$ ($\mid\!\vDash$ G) $\longrightarrow$ ($\vDash$ e : A), where $\mid\!\vDash$ {} is trivial. The key property is soundness of G $\rhd$ e : A:

[1]The VDM logic is mechanised by Hans-Wolfgang Loidl, and the integration is joint work with him. In this section, he has conducted the work purely on the expression layer.

theorem `soundness`: $\mathsf{G} \rhd \mathsf{e} : \mathsf{A} \Longrightarrow \mathsf{G} \vDash \mathsf{e} : \mathsf{A}$

Finally, note that *validity* ($\vDash \mathsf{e} : \mathsf{A}$) is defined as:

$$\vDash \mathsf{e} : \mathsf{A} \equiv (\forall\ \mathsf{E}\ \mathsf{h}\ \mathsf{hh}\ \mathsf{v}\ \mathsf{p}\ .\ (\mathsf{E},\ \mathsf{h} \vdash \mathsf{e} \Downarrow (\mathsf{v},\mathsf{hh},\mathsf{p})) \longrightarrow \mathsf{A}\ \mathsf{E}\ \mathsf{h}\ \mathsf{hh}\ \mathsf{v}\ \mathsf{p})$$

9.2.2 Integrating the TLA and VDM-style embeddings

In the integration, the expression layer is represented by $\mathsf{E},\ \mathsf{h} \vdash \mathsf{e} \Downarrow (\mathsf{v},\mathsf{hh},\mathsf{p})$, and the required properties are proved by $\{\} \rhd \mathsf{e} : \mathsf{A}$. The coordination layer is represented in TLA using types HoVal and Heap. e will update the state, and A (from $\{\} \rhd \mathsf{e} : \mathsf{A}$) will capture required properties of e, used in the TLA proofs of coordination layer properties. The lifted TLA logic, together with the shallow VDM assertions, ease the integration. However, to enable integration, e must be represented as a function, which cannot be directly achieved due the deep embedding of e. Thus, a function exe is defined. It accepts an environment E, a heap h and an expression e, and returns a triple consisting of a value v, a new heap hh and resource consumed p. This is axiomatised with respect to the semantic definition of e:

axioms exe: $(\mathsf{exe}\ \mathsf{E}\ \mathsf{h}\ \mathsf{e} = (\mathsf{v},\mathsf{hh},\mathsf{p})) = (\mathsf{E},\ \mathsf{h} \vdash \mathsf{e} \Downarrow (\mathsf{v},\mathsf{hh},\mathsf{p}))$

vdmexe is then used to connect exe, $\{\} \rhd \mathsf{e} : \mathsf{A}$ and the desired property as a HOL predicate, which is used in a TLA proof of coordination layer properties:

theorem `vdmexe`: $[\![\ (\mathsf{v},\mathsf{hh},\mathsf{p}) = \mathsf{exe}\ \mathsf{E}\ \mathsf{h}\ \mathsf{e};\ \{\} \rhd \mathsf{e} : \mathsf{A}\ ]\!] \Longrightarrow \mathsf{A}\ \mathsf{E}\ \mathsf{h}\ \mathsf{hh}\ \mathsf{v}\ \mathsf{p}$

Proof outline. The assumption applied to the soundness theorem implies $\{\} \vDash \mathsf{e} : \mathsf{A}$. By the definition of *validity in context*, this implies $(\mid\!\vDash \{\}) \longrightarrow (\vDash \mathsf{e} : \mathsf{A})$ which implies (G1) $\vDash \mathsf{e} : \mathsf{A}$. Moreover, the definition of exe and the assumption of vdmexe, implies $\mathsf{E},\ \mathsf{h} \vdash \mathsf{e} \Downarrow (\mathsf{v},\mathsf{hh},\mathsf{p})$. The goal then follows from unfolding (G1), by using the definition of *validity*, followed by an application of it. $\therefore$

9.2.3 An example

The integration is illustrated by the even-odd example from Section 5.5.2, consisting of the two communicating `even` and `odd` boxes. The embedding of Section 5.5.2 is changed by adding a heap of type Heap statefun. Moreover, the box state, scheduler and program counter are unchanged, while the result buffers and wires are of type:

w1, w2, even_res, odd_res :: HoVal statefun.

Since both the even and odd boxes have the same expression layer, they are represented as the expression body. It accepts an environment E with a mapping from variable x to a HoVal of type integer, i.e. the constructor I i, where i is of type int. The TLA embedding is independent of how body is implemented. All that is required is the theorem

theorem body1: $\{\} \rhd \text{body} : \lambda\ \text{E h hh v p.}\ \forall\ \text{i r.}\ \text{E x} = (\text{I i}) \longrightarrow \text{v} = (\text{I (i+1)})$.

To simplify the TLA encoding, a function exebody is then defined as

exebody v h ≡ exe emptyEnv(x := v) h body.

The overall structure of the execute phase of each box is similar to Section 4.5: a test on the program counter followed by a check on the box state. Following the Hume semantics [112], patterns are first checked for matches, and the expression layer is only called if a match succeeds. Here, this means that the input wire is not empty (#⊥). If this is empty, everything are unchanged. If not, the wire is consumed and exebody is called[2]:

```
if $w2 = #⊥
 then (even_res,even_st,w2,heap)$ = (#⊥,#Matchfail,$w2,$heap)
 else (w2,even_st)$ = #(⊥,Runnable) ∧ (even_res,heap,rsrc)$ = exebody<$w2,$heap>
```

The odd_exe action is similar. The super-step can no longer be atomic and distributed among the boxes, since the same heap is updated by all boxes. Distributing it will now require a lower abstraction level, where the super-step phase contains several sequential (TLA) steps. Instead, one specific action that super-steps all boxes is defined. The initial state init and full program specification program are similar to Section 4.5.

The same property as in Section 4.5 is verified, however here the two sets Even and Odd for integers replaces EvenN and OddN for naturals. Now, exebody1 joins the body1 and vdmexe theorems with the definition of exebody, while l1 shows that the wire is either empty or an integer:

lemma exebody1: $(\text{vv,hh,st}) = \text{exebody (I n) h} \Longrightarrow \text{vv} = \text{I (n+1)}$

lemma l1: $\vdash \text{program} \longrightarrow \Box((\exists\ \text{n.}\ \$\text{w1} = \#(\text{I n})) \vee \$\text{w1} = \#\bot)$
$\wedge\ (\exists\ \text{n.}\ \$\text{w2} = \#(\text{I n})) \vee \$\text{w2} = \#\bot)$
$\wedge\ (\exists\ \text{n.}\ \$\text{even\_res} = \#(\text{I n})) \vee \$\text{even\_res} = \#\bot)$

[2] rsrc contains the information about resources and is not used in this program.

$$\wedge\ (\exists\ \mathsf{n}.\ \$\mathsf{odd\_res} = \#(\mathsf{I}\ \mathsf{n}))\ \vee\ \$\mathsf{odd\_res} = \#\bot))$$

Proof outline. The proof of exebody1 follows directly from the definition of exebody and Theorem body1 and the vdmexe theorem. The proof of I1 follows the structure of a TLA invariant, with case analysis on the scheduler, the program counter, the box states and pattern matching. The main part of the proof requires body1 and is mainly by correct instantiation of bound variables. $\therefore$

theorem `main`:

$$\begin{aligned}\vdash \mathsf{program} \longrightarrow\ & \Box((\forall\ \mathsf{n}.\ \$\mathsf{w1} = \#(\mathsf{I}\ \mathsf{n})) \longrightarrow \#\mathsf{n} \in \#\mathsf{Even}) \\ & \wedge\ (\forall\ \mathsf{n}.\ \$\mathsf{w2} = \#(\mathsf{I}\ \mathsf{n})) \longrightarrow \#\mathsf{n} \in \#\mathsf{Odd}) \\ & \wedge\ (\forall\ \mathsf{n}.\ \$\mathsf{even\_res} = \#(\mathsf{I}\ \mathsf{n})) \longrightarrow \#\mathsf{n} \in \#\mathsf{Even}) \\ & \wedge\ (\forall\ \mathsf{n}.\ \$\mathsf{odd\_res} = \#(\mathsf{I}\ \mathsf{n})) \longrightarrow \#\mathsf{n} \in \#\mathsf{Odd}))\end{aligned}$$

Proof outline. The proof structure is similar to the proof of Lemma I1, and uses the lemmas I1 and exebody1. $\therefore$

The proof that $\Box(\forall\ \mathsf{n}.\ \$\mathsf{w1} = \#(\mathsf{I}\ \mathsf{n})) \longrightarrow \#\mathsf{n} \in \#\mathsf{Even})$ and $\Box(\forall\ \mathsf{n}.\ \$\mathsf{w2} = \#(\mathsf{I}\ \mathsf{n})) \longrightarrow \#\mathsf{n} \in \#\mathsf{Odd})$ follows by applying (STL5) to Theorem main.

9.2.4 Discussion

This section has illustrated how the TLA and VDM techniques can be integrated into a proof environment for Hume. Moreover, the actual definitions are independent of each other, since the integration only depends on the proved theorems (e.g. body1). Section 5.5.1 illustrated how TLA (TLA$^+$) could be used to model check real-time properties. Such real time properties, as well as space properties, should be possible to verify in this VDM/TLA embedding. Another future work is to derive the exe axiom properly. Now, the main problem is the heap representation. The formal semantics define a wire heap and one separate heap for each box. This involves a very costly deep copy between the wire heap and box heaps, and the cost model does not capture this. This example used one heap, which was not accessed by any variables since dynamic types, like lists, were not used. A similar one heap approach, was suggested by Vasconcelos' PhD thesis [189]. Here, heap regions was used. However, for correctness properties a single heap representation, require a proof that a box does not change the part of the heap "owned" by other boxes. An elegant solution to capture this, is to introduce the *frame rule* of separation logic [170], into the VDM-logic.

9.3 Liveness

9.3.1 Liveness in flat Hume

Liveness in TLA was introduced in Section 2.3.1. Here, it is discussed for the abstract TLA formalisation of Hume shown in Figure 4.2, although the sequential embedding can be updated similarly. Moreover, as with the TLA proof rules, the focus is on leads-to properties $P \leadsto Q$. Now, liveness properties cannot be proved from the specification shown in Figure 4.2, since $\Box[\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i]_{\langle s,ws,res,st \rangle}$ does not rule out the infinite sequence of stuttering steps:

$$\big\langle \langle s,ws,res,st \rangle, \langle s,ws,res,st \rangle, \langle s,ws,res,st \rangle, \langle s,ws,res,st \rangle, \cdots \big\rangle,$$

where $\langle s,ws,res,st \rangle$ is the initial value of these state components. To prove liveness properties, the specification must be constrained by liveness assumptions, and as explained in Section 2.3.1, these constraints should take the form of a *fairness* assumption. Moreover, these must be sub-actions of $\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i$ to ensure machine closure [5, page 519], i.e. additional safety constraints are not introduced. Fairness assumptions on $\mathcal{S}$, $\mathcal{B}_i$ (for all $i \in BS$) and $\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i$ fall into this category and are implied by the Hume semantics. Moreover, all these actions are always enabled, albeit this must be formally verified. Assuming this, then by applying standard temporal reasoning,

$$\Box \mathit{Enabled}\langle \mathcal{S} \rangle_{\langle s,ws,res,st \rangle} \Rightarrow \Diamond\Box \mathit{Enabled}\langle \mathcal{S} \rangle_{\langle s,ws,res,st \rangle} \Rightarrow \Box\Diamond \mathit{Enabled}\langle \mathcal{S} \rangle_{\langle s,ws,res,st \rangle}$$

can be deduced. Since, $\mathcal{S}$ is always enabled, and by the definition of SF and WF $\big($(2.3) on page 15$\big)$, H_l implies $SF_{\langle s,ws,res,st \rangle}(\mathcal{S}) \equiv \Box\Diamond\langle \mathcal{S} \rangle_{\langle s,ws,res,st \rangle}$ and $WF_{\langle s,ws,res,st \rangle}(\mathcal{S}) \equiv \Box\Diamond\langle \mathcal{S} \rangle_{\langle s,ws,res,st \rangle}$, which means that H_l implies

$$SF_{\langle s,ws,res,st \rangle}(\mathcal{S}) \equiv WF_{\langle s,ws,res,st \rangle}(\mathcal{S}). \tag{9.1}$$

The same can be deduced for $\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i$ and all the $\mathcal{B}_i$ actions. Thus, using these, both weak and strong fairness can be used to prove $P \leadsto Q$, which means applying (WF1) or (SF1) respectively. Only weak fairness is discussed further. For a machine closed action $\mathcal{A}$ of a TLA embedded Hume program, (WF1) is instantiated as

$$\frac{\begin{array}{l} \vdash P \wedge [\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i]_{\langle s,ws,res,st \rangle} \Rightarrow (P' \vee Q') \\ \vdash P \wedge \langle \mathcal{S} \wedge \big(\bigwedge_{i \in BS} \mathcal{B}_i\big) \wedge \mathcal{A} \rangle_{\langle s,ws,res,st \rangle} \Rightarrow Q' \\ \vdash P \Rightarrow \mathit{Enabled}\ \langle \mathcal{A} \rangle_{\langle s,ws,res,st \rangle} \end{array}}{\vdash \Box[\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i]_{\langle s,ws,res,st \rangle} \wedge WF_{\langle s,ws,res,st \rangle}(\mathcal{A}) \Rightarrow (P \leadsto Q).}$$

Here, line 1 shows that if P holds then either P or Q must hold after both if the subscript is left unchanged and if the next action is applied. Line 2 states, executing $\mathcal{A}$, implies that Q holds afterwards, while line 3 states that $\mathcal{A}$ must be weak enough such that P implies that it is enabled. To illustrate, $\mathcal{A}$ is instantiated to $\mathcal{S}$, to prove that $(s = \mathit{Execute}) \rightsquigarrow (s = \mathit{Super})$:

$$\begin{array}{l} 1.\ \vdash s = \mathit{Execute} \wedge [\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i]_{\langle s, ws, res, st \rangle} \Rightarrow (s' = \mathit{Execute} \vee s' = \mathit{Super}) \\ 2.\ \vdash s = \mathit{Execute} \wedge \langle \mathcal{S} \wedge \big(\bigwedge_{i \in BS} \mathcal{B}_i \big) \wedge \mathcal{S} \rangle_{\langle s, ws, res, st \rangle} \Rightarrow s' = \mathit{Super} \\ 3.\ \vdash s = \mathit{Execute} \Rightarrow \mathit{Enabled}\ \langle \mathcal{S} \rangle_{\langle s, ws, res, st \rangle} \\ \hline \vdash \Box[\mathcal{S} \wedge \bigwedge_{i \in BS} \mathcal{B}_i]_{\langle s, ws, res, st \rangle} \wedge \mathit{WF}_{\langle s, ws, res, st \rangle}(\mathcal{S}) \Rightarrow (s = \mathit{Execute} \rightsquigarrow s = \mathit{Super}) \end{array}$$

Proof outline. Line 1 states that by applying the next state action, or leaving the state space unchanged s either remains *Execute* or becomes *Super*. This trivially holds by $\mathcal{S}$. Line 2 states that executing the next action (and $\mathcal{S}$) such that the state space changes, forces s to become *Super*, while line 3 asserts that $\mathcal{S}$ is enabled when $s = \mathit{Execute}$. Both of these trivially hold. $\therefore$

The $\rightsquigarrow$ operator is transitive (Lemma LT13 of Appendix A.1). Reasoning about a $P \rightsquigarrow Q$ correctness property, will most likely require reducing this to smaller subgoals:

$$(P \rightsquigarrow R_1) \wedge (R_1 \rightsquigarrow R_2) \wedge \cdots \wedge (R_n \rightsquigarrow Q).$$

For example, let w_i be an input wire, and w_o an output wire of box $i \in BS$, where p projects the value of res_i connected to w_o. To verify $P(w_i) \rightsquigarrow Q(w_o)$, the proof can be reduced to $P(w_i) \rightsquigarrow Q\big(p(res_i)\big)$ and $Q\big(p(res_i)\big) \rightsquigarrow Q(w_o)$. Moreover, for a "coordination iteration" loop, as in Section 7.2, the (LATTICE) rule can be applied to prove "termination" if the type is well-founded and always reducing.

For example, to verify $P(w_i) \rightsquigarrow Q\big(p(res_i)\big)$, s must be *Execute* and st_i cannot be *Blocked*. Moreover, the fairness constaint $\mathcal{A}$ must imply that with these assumptions, together with $P(w_i)$, $Q\big(p(res_i')\big)$ holds as a result. Thus, it may be necessary to strengthen the fairness constraint to $(s = \mathit{Execute} \wedge \mathcal{B}_i^e)$ and $(s = \mathit{Super} \wedge \mathcal{B}_i^s)$ for all $i \in BS$, albeit this requires more research. Moreover, to show $Q\big(p(res_i)\big) \rightsquigarrow Q(w_o)$, is must be shown that w_o is empty. This requires a proof that the destination box of w_o is not *Blocked*. This proof may require a proof that other boxes are not *Blocked*, and so on. This may even depend on the *Blocked* status of the other boxes in the program.

Model checking example: revisiting the traffic lights[3]

Liveness for model checking is illustrated by updating the traffic light example from Section 5.5.1 with liveness. Moreover, $\Box\Diamond P$ properties, which can be writeen $True \leadsto P$, are discussed. It is sufficient with weak fairness for the complete next-state action, thus (5.5) is updated to:

$$Init \wedge Init_{tl_1,tl_2} \wedge \Box[\mathcal{N} \wedge \mathcal{S} \wedge \mathcal{E} \wedge \mathcal{N}_{tl_1,tl_2}]_{\langle tl_1,tl_2,v\rangle} \wedge WF_{\langle tl_1,tl_2,v\rangle}(\mathcal{N} \wedge \mathcal{S} \wedge \mathcal{E} \wedge \mathcal{N}_{tl_1,tl_2}).$$

The first liveness property states that at any given time, there will always be a time in the future when the lights are green:

$$\Box\Diamond\big(tl_1 = (0,0,1)\big) \wedge \Box\Diamond\big(tl_2 = (0,0,1)\big)$$

An example of a leads-to property is that if tl_1 is red then tl_2 will eventually become green, which is specified as

$$\big(tl_1 = (1,0,0) \leadsto tl_2 = (0,0,1)\big) \wedge \big(tl_2 = (1,0,0) \leadsto tl_1 = (0,0,1)\big)$$

9.3.2 Liveness in Hierarchical Hume

The dependency liveness properties have on the *Blocked* status of, potentially, all boxes in a program, is another example of the advantage of the structuring introduced by Hierarchical Hume. Here, this dependency can be reduced and controlled. However, while first-level boxes may have the same fairness assumptions as in flat Hume, execution of nested boxes depends on the parent. Now, since nested boxes execute when the parent's state is *Execute* or *Super*, a possible fairness constraint is $WF_{\langle s,ws,res,st\rangle}(boxsch(i) \in \{Execute, Super\} \Rightarrow \mathcal{B}_i)$, but this requires further research. Note that termination of a nesting box can be expressed as $st_i \in \{Execute, Super\} \leadsto st_i = Terminated$.

When constraining a program with fairness, the fairness constraints must be preserved by a transformation, and thus also incorporated into the box calculus. This is verified by (WF2) and (SF2), for weak and strong fairness respectively. Note that, as shown in the definition of (WF2) and (SF2), refinement mapping (i.e. substitutions) does not distribute over *Enabled*. The verification of transformation of programs with liveness constraints is more difficult than the safety aspect, and requires further study.

[3]This example is published in [91]

```
mult: inp1 := w1; inp2 := w2;
      { True }
        a := inp1; b := inp2;
        r := 0; x := a; y := b;
        while (y ≠ 0)
        begin
          r := r + x;   y := y − 1;
        end
        o := r;
        out := o;
      {out = inp1 ∗ inp2 }
```

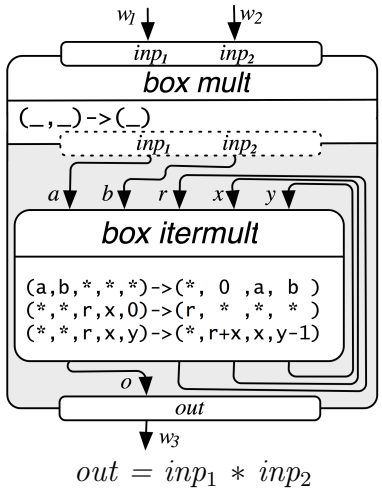


Figure 9.2: Left: imperative multiplication as iteration. Right: Hume multiplication as iteration

9.3.3 Discussion

The Hume coordination layer possesses many interesting liveness properties which TLA is well equipped to handle. This section has discussed fairness constraints of the Hume coordination layer, illustrated the proof of a small scheduling liveness property, and shown liveness verification by model checking. Fairness constraints for Hierarchical Hume have also been briefly discussed. The liveness work is still at a very early stage of development, and liveness is, in general, considered much harder than safety. However, as with safety properties, by exploring the strict structure and scheduling of Hume programs, it may be possible to implement tactics to automate liveness proofs as well.

9.4 Towards property discovery automation[4]

This section discusses the use of *rippling* [33] and *proof critics* [103] to discover (D3) type invariants and verify transformations. It is achieved by adapting previous work with imperative programs [105, 106] into TLA-embedded Hierarchical Hume programs. The mult box from Section 7.2 is used as illustration, however the result buffer and scheduling phases of the nested box are abstracted over to ease the reading. This is shown on the right hand side of Figure 9.2, while the left hand side shows the corresponding Hoare annotated imperative code.

[4]This section has been published in [86]

9.4.1 Rippling background

Input sequent: $H \vdash G[f_1(\boxed{c_1(\ldots)}^{\uparrow}, f_2(\lfloor \ldots \rfloor), f_3(\boxed{c_2(\ldots)}^{\uparrow}))]$

Method preconditions:

1. there exists a subterm T of G which contains wave-front(s), *e.g.* $f_1(\boxed{c_1(\ldots)}^{\uparrow}, f_2(\lfloor \ldots \rfloor), f_3(\boxed{c_2(\ldots)}^{\uparrow}))$
2. there exists a wave-rule which matches T, *e.g.*
   $C \rightarrow f_1(\boxed{c_1(\boxed{X})}^{\uparrow}, Y, Z) \Rightarrow \boxed{c_5(f_1(X, \boxed{c_3(Y)}^{\downarrow}, \boxed{c_4(Z)}^{\downarrow}))}^{\uparrow}$
3. the wave-rule condition follows from the context, *e.g.* $H \vdash C$
4. resulting inward directed wave-fronts are potentially removable, *e.g. sinkable or cancellable, i.e.*
   $\ldots \boxed{c_3(f_2(\lfloor \ldots \rfloor))}^{\downarrow} \ldots$ or $\ldots \boxed{c_4(f_3(\boxed{c_2(\ldots)}^{\uparrow}))}^{\downarrow} \ldots$

Output sequent: $H \vdash G[\boxed{c_5(f_1(\ldots, \boxed{c_3(f_2(\lfloor \ldots \rfloor))}^{\downarrow}, \boxed{c_4(f_3(\boxed{c_2(\ldots)}^{\uparrow}))}^{\downarrow}))}^{\uparrow}]$

Figure 9.3: The rippling method [33]

Rippling [33] is a proof planning [32] rewriting technique, based upon a difference reduction strategy. Rippling can be illustrated by considering a conjecture with a hypothesis of the form $(\forall b'.\ f(a, b'))$ and a goal of the form $f(c_1(a), b)$. Here, the $c_1(\ldots)$ embedded within the goal prevents a match with the hypothesis. In rippling, such embedded structures are called *wave-fronts*. The goal of rippling is to identify and reduce the number of wave-fronts such that a hypothesis can be applied. Wave-fronts can be represented using explicit annotations that are added to the goal. For example, using shading to denote wave-fronts, the goal given above becomes $f(\boxed{c_1(\boxed{a})}^{\uparrow}, \lfloor b \rfloor)$. In addition to the shading, note that a wave-front is annotated with an arrow. The arrow indicates which direction the wave-front can be moved, *i.e.* upward or downward through the goal structure. There are two reasons for moving a wave-front downward. Firstly, if a wave-front can be moved to a position corresponding to a universally quantified variable within the given hypothesis, then the wave-front can be eliminated via the specialisation of the universal hypothesis. This is known as *sinking* a wave-front, and the $\lfloor \ldots \rfloor$ annotation within the goal is used to indicate sink positions. Secondly, multiple wave-fronts can sometimes cancel each other out, a kind of destructive inter-

ference. So moving wave-fronts closer together can also make sense. The manipulation of wave-fronts is achieved via *wave-rules*, which are rewrite rules that have been annotated by wave-fronts. Wave rules must preserve the unannotated structure of the goal, known as the *skeleton*. This preservation maximises the chances of eventually applying the given hypothesis. In the schematic example given above, the following represents an applicable wave-rule:

$$f(\boxed{c_1(\boxed{X})}^{\uparrow}, Y) \Rightarrow \boxed{c_2(\boxed{f(X, \boxed{c_3(\boxed{Y})}^{\downarrow})})}^{\uparrow}$$

Now, a proof plan contains *methods* and *critics* [102]. Methods represent common patterns of reasoning, and Figure 9.3 shows the rippling method. Critics are used to define patchable exceptions. When a method fails, its associated critics analyse the proof-failure and initiate a proof patch [102, 103, 104]. Typically the proof patching process uses meta-variables as place-holders for missing structure. Here, it is expected that the constraints of the proof will provide instantiations during the planning of the remainder of the proof. This style of patching a proof is known as *middle-out reasoning* [34], and is used in the proof plans below.

9.4.2 Invariant verification

In [105], rippling was used to verify Hoare-triple properties as illustrated on the left hand side of Figure 9.2. To verify a Hoare-triple, it is converted into a *verification condition* (VC), a purely logical statement, by a *verification condition generator* (VCG). The VC is then verified by a theorem prover. However, before this can be done, each statement must be turned into a Hoare-triple. This is mostly an automatic process, however finding and verifying an invariant which holds throughout the ***while*** loop, known as the *loop invariant*, is the most difficult aspect. Thus, loop invariants are focused upon here. In the program on the left hand side of Figure 9.2 the loop invariant is $r + (x * y) = inp_1 * inp_2$. In the proof, the invariant is assumed beforehand, and this assumption is called the *invariant hypothesis* (IH)

$$\mathsf{IH}: \; r + (x * y) = inp_1 * inp_2. \tag{9.2}$$

IH is assumed before the loop execution, and it is showed that it holds afterwards. By using wave-annotation, this goal is expressed as $\boxed{(\boxed{r} + x)}^{\uparrow} + (x * \boxed{(\boxed{y} - 1)}^{\uparrow}) = inp_1 * inp_2$. The proof requires the following wave-rules:

$$(X + Y)^{\uparrow} + Z \Rightarrow X + (Y + Z)^{\downarrow} \tag{9.3}$$

$$X * (Y - 1)^{\uparrow} \Rightarrow (X * Y) - X^{\uparrow} \tag{9.4}$$

$$(X + (Y - X^{\uparrow})^{\downarrow}) \Rightarrow Y, \tag{9.5}$$

and is derived as follows:

$$\begin{array}{ll}
(r + x)^{\uparrow} + (x * (y - 1)^{\uparrow}) = inp_1 * inp_2 & \text{[apply (9.3)]} \\
r + (x + (x * (y - 1)^{\uparrow}))^{\downarrow} = inp_1 * inp_2 & \text{[apply (9.4)]} \\
r + (x + ((x * y) - x^{\uparrow}))^{\downarrow} = inp_1 * inp_2 & \text{[apply (9.5)]} \\
r + (x * y) = inp_1 * inp_2 & \text{[apply IH]}
\end{array}$$

The last match of `itermult` responds to the ***while*** loop in the imperative program. Since the match expression is the result of executing a Hume box, it refers to the primed result state of an action. The annotated result of translating `(*,r-x,x,y-1)` into TLA becomes:

$$r' = (r + x)^{\uparrow} \quad x' = x \quad y' = (y - 1)^{\uparrow} \quad inp'_1 = inp_1 \quad inp'_2 = inp_2. \tag{9.6}$$

The "loop invariant" in TLA is the same as in the imperative program, and the IH for the invariant proof is also the same (IH: $r + (x * y) = inp_1 * inp_2$). The ripple proof derivation starts off as

$$\begin{array}{ll}
r' + (x' * y') = inp'_1 * inp'_2 & \text{[apply (9.6)]} \\
(r + x)^{\uparrow} + (x * (y - 1)^{\uparrow}) = inp_1 * inp_2 & [\cdots]
\end{array}$$

and the remaining proof is identical to the imperative program. Note, that the first annotation step here is handled by the VCG in the imperative version.

9.4.3 Loop invariant discovery

The most difficult aspect of Hoare-triple proofs, is the undecidable task of finding a strong enough loop invariant, like (9.2). [105] contains novel work, where proof critics [103] are used to explore partial ripple successes to discover a strong enough loop

Blockage: $\boxed{r + x}^{\uparrow} = inp_1 * inp_2$

Critic precondition:

- Precondition 1 of rippling succeeds, *i.e.*
  1. there is a subterm T of G which contains a wave-front(s), e.g. $\boxed{r + x}^{\uparrow}$
- Precondition 2 of rippling partially succeeds, *i.e.*
  2. there exists a wave-rule which partially matches, e.g. $\boxed{r + x}^{\uparrow}$
  with $\boxed{(X + Y)}^{\uparrow} + Z \Rightarrow \ldots$

Proof patch:

Speculate additional term structure within the conjecture such that preconditions 2, 3 and 4 will also potentially succeed, *i.e.* $F_1(\boxed{r + x}^{\uparrow}, x, \boxed{y - 1}^{\uparrow}) = inp_1 * inp_2$, where F_1 is a higher-order meta-variable.

Figure 9.4: A tail-invariant proof critic instantiation (adapted from [105])

invariant. Firstly, both a ***while*** loop and a Hume feedback loop, as in Figure 9.2, require a tail-invariant, and the proof critic thus provides a tail-invariant patch. We will now apply the work described in [105] to discover the feedback loop Hume invariant required for the proof in the previous section. Now, the post-condition for the property is $out = inp_1 * inp_2$, which is updated as follows: $out' = o$ and $o' = r$. Thus, an obvious first approximation of the loop invariant becomes:

$$\mathsf{IH}: \; r = inp_1 * inp_2.$$

Now, by the TLA "induction rule" and (9.6), the proof of the "loop action" blocks when attempting to ripple

$$\underbrace{\boxed{r + x}^{\uparrow}}_{\textbf{blocked}} = inp_1 * inp_2.$$

where blocking means that no wave rule applies, hence rippling is not possible. However, the precondition of the tail-invariant proof critic succeeds, as illustrated in Figure 9.4. Thus, this partial match can be explored and the patch suggests a schematic invariant of the form

$$F_1(r, x, y) = inp_1 * inp_2, \tag{9.7}$$

where F_1 is a second-order meta-variable. The expectation is that F_1 will be instantiated during the course of the proof. The primed variables in $F_1(r', x', y') = inp'_1 * inp'_2$ are first rewritten using (9.6)

$$F_1(\boxed{r + x}^{\uparrow}, x, \boxed{y - 1}^{\uparrow}) = inp_1 * inp_2.$$

By wave-rule (9.3), a new meta-variable F_2 is introduced by instantiating F_1 to $\lambda X.\lambda Y.\lambda Z.X + F_2(X, Y, Z)$. Application of (9.3) then derives

$$r + \boxed{x + F_2(\boxed{r + x}^{\uparrow}, x, \boxed{y - 1}^{\uparrow})}^{\downarrow} = inp_1 * inp_2.$$

Here, (9.4) suggests a new variable F_3, such that F_2 is instantiated to $\lambda X.\lambda Y.\lambda Z.F_3(X, Y, Z) * (Z - 1)$. The result of applying (9.4) is then

$$r + \boxed{x + \boxed{F_3(\boxed{r + x}^{\uparrow}, x, \boxed{y - 1}^{\uparrow}) * y - F_3(\boxed{r + x}^{\uparrow}, x, \boxed{y - 1}^{\uparrow})}^{\uparrow}}^{\downarrow}$$

Finally, wave rule (9.5) suggests that F_3 is instantiated to $\lambda X.\lambda Y.\lambda Z.Y$, resulting in the following invariant:

$$r + (x * y) = inp_1 * inp_2.$$

The invariant is identical to the invariant in Section 9.4.2, and the proof structure is the same.

9.4.4 Hume program transformations verification

Here, the invariant discovery is extended to Hume program transformations. This is illustrated with the transformation verified in Section 7.2, where the `emult` box is transformed into the `mult` box. Section 6.6 showed that such transformations reduce to the partial correctness property:

$$out = \mathsf{fmult}\ 0\ inp_1\ inp_2. \tag{9.8}$$

Thus, as in the previous sections, the key to the proof of (9.8), is the loop invariant of the nested feedback loop of mult. Now, from the definition of `fmult` the following conditional wave-rule is derived:

$$Y \neq 0 \rightarrow \mathsf{fmult}\ \boxed{(\boxed{R} + X)}^{\uparrow}\ X\ \boxed{(\boxed{Y} - 1)}^{\uparrow} \Rightarrow \mathsf{fmult}\ R\ X\ Y \tag{9.9}$$

This is the only rule required in the proof. The loop invariant for this proof is $\mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2$, and the induction hypothesis is obviously the same:

$$\mathsf{IH}\text{: } \mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

The "loop step" of the `mult` then induces the following derivation:

$$\begin{array}{rl}
\mathsf{fmult}\ r'\ x'\ y' = \mathsf{fmult}\ 0\ inp'_1\ inp'_2 & [\mathsf{apply\ (9.6)}] \\
\mathsf{fmult}\ \boxed{(\boxed{r} + x)}^{\uparrow}\ x\ \boxed{(\boxed{y} - 1)}^{\uparrow} = \mathsf{fmult}\ 0\ inp_1\ inp_2 & [\mathsf{apply\ (9.9)}] \\
\mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2 & [\mathsf{apply\ IH}]
\end{array}$$

Note that in the application of (9.9), the pre-condition of the rule holds, by the definition of the corresponding pattern. If `y = 0`, the second match would have succeeded.

Invariant discovery

The tail-invariant critic, where one particular instantiation is illustrated in Figure 9.4, can be reused to discover the loop invariant in this example, although the particular rules will defer. Similarly to Section 9.4.3, the first approximation of the invariant is $r = \mathsf{fmult}\ 0\ inp_1\ inp_2$, which forces rippling to block:

$$\underbrace{\boxed{\boxed{r} + x}^{\uparrow}}_{\textbf{blocked}} = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

However, the precondition of the tail-invariant patch succeeds, which results in the introduction of a meta-variable F_1:

$$F_1(r, x, y) = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

The definition of the priming operators (9.6) results in:

$$F_1(\boxed{\boxed{r} + 1}^{\uparrow}, x, \boxed{\boxed{y} - 1}^{\uparrow}) = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

Wave-rule (9.9) then suggests instantation F_1 directly to fmult, and the same loop invariant as in the previous section is obtained:

$$\mathsf{fmult}\ r\ x\ y = \mathsf{fmult}\ 0\ inp_1\ inp_2$$

9.4.5 Discussion

The work presented here builds directly on [105, 106], which use rippling [33] and proof critics [103] to verify imperative programs. Here, this work is reused in the Hume/TLA context, and extended to verify Hume program transformations. The work in [105, 106] has been implemented and the Hierarchical Hume examples have been verified in Section 7.2. Thus, since Isabelle/HOL supports rippling via IsaPlanner [59, 57, 58], implementation should not be too involved.

9.5 Summary & discussion

In this chapter three explorations, relevant to the critical path of the thesis, are explored. They have shown that the 'integration with expression layer' and 'liveness extensibility' motivations behind TLA and theorem proving within Isabelle/HOL are sound. It has also been shown that automations within the current mechanisation may be possible by using rippling and proof critics in IsaPlanner.

Chapter 10

Future work & conclusion

10.1 Contributions revisited

The main contributions of this thesis, listed in Chapter 1, have been implemented as follows:

- **correctness and transformation verification of Hume programs**. The thesis is the first to discuss verification of correctness properties and transformations of Hume programs. This is achieved using both model checking and theorem proving, and tactics are developed to automate the theorem proving aspect. Chapter 5 discusses Hume reasoning, illustrated by case-studies, while Chapter 6 discusses verification in the Hierarchical Hume extension. This is illustrated by case-studies in Chapter 7. In [85], several verification approaches for Hume, including TLA, is outlined;

- **TLA is used at the programming language level and in program transformation verification.** This work is novel in applying TLA to the programming language level. Chapter 4 formalises Hume in TLA, and mechanises Hume in TLA$^+$ (TLC) and Isabelle/TLA, while Chapter 6 formalises Hierarchical Hume in TLA, and mechanises it in Isabelle/TLA. Compared to more abstract specification, the programming language level is more concrete and specific, thus enables the development of tactics in Chapters 5 and 6 to automate the verification. However, it often requires more proofs, as illustrated in the SAFER case study in Chapter 7 which has also been embedded as a more abstract model in PVS;

- **TLA is mechanised in the Isabelle/HOL theorem prover.** This mechanisation is achieved in Chapter 3, and is required for the deductive Hume verification in subsequent chapters. TLA has previously been mechanised [65, 115, 128,

143, 197, 202], however this mechanisation has several contributions. Particularly, is the formalisation of sequences which enables proofs of stuttering invariance, the embedding of TLA*, proof of stuttering invariance of TLA* operators, and the derivation of the TLA* proof system and higher-level TLA rules;

- **Hume is formalised in TLA, mechanised in the TLC model checker and Isabelle/TLA, and tactics developed.** From the Hume perspective, the formalisation and mechanisations in Chapter 4 are major contributions. This is also the case for the informal proof plans, implemented as tactics in Chapter 5. The tactics achieved a high automation when the property specified is sufficiently strong. [91] discusses verification of HW-Hume programs using TLC;
- **the development, formalisation and mechanisation of Hierarchical Hume.** In joint work with Robert Pointon, Hierarchical Hume has been developed in this thesis. It has been formalised in TLA, and mechanised in Isabelle/TLA. Informal proof plans for both invariant and transformation verification have been developed, and implemented as proof tactics in Chapter 6. [89] motivates and illustrates Hierarchical Hume in the context of transformations;
- **the verification of scheduling strategies.** In Chapter 4, the existing self-out scheduling extension was shown to be conservative with respect to the standard Hume lock-step scheduling strategy. In Chapter 6, the Hierarchical Hume extension was shown to be conservative. Both these properies were verified using TLA, and the complete detailed proofs are shown in Appendix B. [88] discusses verification of scheduling algorithms using TLA, illustrated in Hume;
- **case studies.** Empirical evidence for both the model checking and theorem proving approach for Hume verification is provided through case studies in Chapter 5. However, the major case studies use theorem proving to verify invariants and transformations of Hierarchical Hume programs (Chapter 7). These also show the applicability of Hierarchical Hume as a programming language. This is particularly the case for the main case study, an implementation of NASA's SAFER system [153];
- **an outline of a box calculus for Hume transformations.** A box calculus for transformations is outlined in Chapter 8. Here, transformations are guided, instead of ad-hoc, which may help increase automation. The box calculus is illustrated by examples, and is published in [87];
- **the integration of Isabelle/Hume with expression layer mechanisation is surveyed.** The Isabelle/HOL theorem proving approach is mainly motivated

by a future integration with a mechanisation of the expression layer in parallel [135]. This motivation is justified by such an integration experiment, illustrated by an example (Chapter 9);

- **the use of rippling to automate both correctness and transformation proofs is surveyed.** Chapter 9 shows how an existing rippling approach for imperative programs, can be reused in an Hume/TLA setting. This is used to speculate (D3) style loop invariants, and extended to transformation verification. This is not implemented, however rippling is supported in the Isabelle/HOL framework through IsaPlanner. This has been published in [86];
- **the verification of Hume liveness properties is surveyed.** Although the focus in this thesis is on safety properties, TLA was partly chosen to enable a future liveness extension. Such an extension is briefly surveyed in Chapter 9.

10.2 Limitations

Some topics are not treated in this thesis. Subjects that may have been expected, are listed and justified below:

- the expression layer is not formalised, and instead structures in the underlying logics are used to describe its behaviour. In Isabelle/HOL this is due the mechanisation of the expression layer conducted in parallel, and the integration work in Chapter 9 partly answers this gap;
- the focus is not on model checking, which could have been expected for a finite state automata based language, like the coordination layer. The focus on the theorem proving approach is motivated by a direct integration with the expression layer integration [135] and the support for the higher Hume levels. However, it is believed that a model checking approach is still worth exploring, which is shown in the Hume case studies (Chapter 5);
- the full Hume language is not treated. In particular, a subset of types is used and exceptions are ignored. The type restriction only has effect in the expression layer, while exceptions are ignored to simplify the embedding. This should be added in the future;
- the properties discussed are limited to invariants and transformations, and address only safety. Since safety properties are simpler and more important than liveness, this is the obvious starting point. The liveness extension is surveyed in Chapter 9;

- formal translation rules from Hume into TLA (Isabelle/HOL and TLC) have neither been defined nor implemented. This becomes more valuable once the integration with the expression layer is more mature, to enable proofs within both layers. A specification language at the Hume level should be part of such an implementation;

- here, the semantics of Hume programs are embedded in TLA, and properties are verified by semantical reasoning using the TLA proofs rules. A more common approach, is to represent a programming language in a theorem prover as a direct inductive embedding of the structural operational semantics (SOS). Then, a high-level logic, like a Hoare logic, is built on top of this, as illustrated for Java [162] and C (using separation logic) [184]. This enables syntactic reasoning at the programming language level, via the high-level logic. This is similar to the approach taken in the embedding of the Hume expression layer, where the Hume SOS [112] is first embedded and a VDM logic is built on top of this. However, as illustrated in Section 9.2, the reasoning within this VDM logic is still very much semantic. Now, the soundness of the work in this thesis, relies on the soundness of TLA, and the creation of a "Hume logic" will shift the focus from software verification to meta-properties like soundness proofs. Thus, focusing on syntactical reasoning would most likely have significantly decreased the empirical evidence gained through case-studies. Thus, it is believed that the more pragmatic method here better illustrates the verification approach. Finally, note that the box calculus outlined in Chapter 8, can be seen as a first step towards such syntactic reasoning, since it is at the Hume source code level.

Note that for most of the work here, TLA introduces an unnecessary overhead. The proofs could have been achieved using simpler formalisations like B [8] and Event-B [9]. However, TLA has been used to enable support of future liveness extensions, and is believed to provide more direct support for integration with the expression layer.

10.3 Future work

10.3.1 The TLA mechanisation

A deep mechanisation of TLA in Isabelle/HOL will remove the extra hypothesis required for several of the TLA* proof rules. It will also enable more meta-level proofs, such as stuttering invariance of the full logic. Moreover, an inductive definition of $\vdash$ and $\vdash\!\!\!\sim$ may enable a completeness proof of at least the propositional part of the TLA*

proof system, and allow a derivation of the "TLA deduction theorem". None of these are directly relevant for the Hume work, and have thus not been explored further.

To properly derive state-dependent properties like ∃ and enableness in Isabelle/TLA, a different state representation is required. Moreover, in [5], Abadi and Lamport introduce the $\stackrel{+}{\triangleright}$ operator. Informally, $A \stackrel{+}{\triangleright} B$ denotes that B holds at least one step longer than A, and is used to avoid circular reasoning in assume-guarantee specifications. Mechanising $\stackrel{+}{\triangleright}$ can be particularly useful to reason about hierarchical boxes, since the children are scheduled independently from the rest of the program.

10.3.2 Properties & specification

A deep mechanisation of (Hierarchical) Hume enables meta-reasoning like, for example, the proofs in Appendix B. Moreover, generic Hume theorems can be verified, and explored by the reasoning techniques.

The focus in the thesis has been on safety properties, while real-time properties is explored in Section 5.5.1, and leads-to properties are surveyed in Section 9.3. There are also properties that do not have to be specified, such as the absence of deadlocks and livelocks, and termination of a nesting box i:

(1) Absence of deadlock: $\Box(\exists i \in BS.\ st_i \neq Blocked)$
(2) Absence of deadlock: $\Box\Diamond\langle \exists i \in BS.\ st_i \neq Blocked\rangle_v$
(3) Absence of livelock: $\Box\Diamond\langle s = Super \Rightarrow \exists i \in BS.\ st_i = Runnable\rangle_v$
(4) Termination of nesting box: $st_i \in \{Execute, Super\} \leadsto st_i = Terminated$

Note that (1) is only applicable for a closed system, since consuming an output stream may resolve the deadlock, while livelocks may be desirable for a reactive system, which may only act on user input.

Once the integration with the expression level is mature, a specification language should be developed at the Hume level, and a translator from Hume into the combined Isabelle/HOL representation should be developed. For Hiearchical Hume, it may be interesting to integrate the specification language with the match: for example, $P(inp)$ `->` $Q(inp, res)$ can be written instead of `_ -> _`, whilst treated like `_ -> _` still in the compiler/interpreter.

10.3.3 (Hierarchical) Hume verification automation

The developed tactics are not very fast. Optimisation could address other existing Isabelle/HOL tools like sledgehammer [142]. In Chapters 5 and 6, proof plans were graphically shown for the verification of invariants and transformations. These were implemented as tactics. However, tactics do not capture the high-level structure of the

proofs, and minor changes to the program structure or property structure will cause them to fail. Instead, they can be implemented as high-level *proof plans* [32], which are supported in Isabelle by IsaPlanner [57, 58]. This may be more robust for small changes, thus supporting better reuse.

In Section 9.4 a proof plan called rippling was used to discover (D3) type (loop) invariants, as classified in Section 5.2, and to verify transformations. A next step is to implement this in IsaPlanner. Moreover, it should be investigated if similar techniques are applicable to (D4) style invariants. Rippling is a terminating rewrite system [33]. Thus non-termination of the Hume simplifier, which occurred in one case study (Section 5.5.3), could be overcome by the use of rippling.

The tactics do not handle failure analysis. Firstly, as explained in Section 5.2, (D1) style invariants always require a proof of the corresponding result buffer, and this can be inferred directly without any sort of program analysis. (D2) type invariant requires strengthening and, as discussed in Section 6.5.1, this approach resembles Dijkstra's weakest precondition predicate transformer *wp* [56]. Here, $wp(C, Q)$ transforms the predicate Q on the result state of the atomic command C to the weakest predicate on the before state of C such that Q holds afterwords. In concurrent executions, like the Hume coordination layer, Lamport argues that a proper generalisation of such partial correctness is invariance, and thus introduced the weakest invariant predicate transformer *win* [119]. From the TLA viewpoint, $win(I \wedge \Box[\mathcal{N}]_v, P) = Q$ implies that Q is the weakest predicate which is invariant over $I \wedge \Box[\mathcal{N}]_v$ such that $Q \Rightarrow P$. This is the type of predicate transformation required for (D3) and (D4) type invariants. However, for a (D2) invariant Q becomes unnecessarily "big" which may cause problems and, at least, slow down the proof tactics. Instead, a more "sequential" strengthening is desirable. For example, a weakest pre-invariant predicate transformer *wpin* where $wpin(I \wedge \Box[\mathcal{N}]_v, P) = Q$ implies the weakest invariant such that $I \wedge \Box[\mathcal{N}]_v \Rightarrow \Box Q$ and $I \wedge \Box[\mathcal{N} \wedge Q \wedge Q']_v \Rightarrow \Box P$. Creating such a *wpin* predicate transformer for at least the (Hiearchical) Hume context is a subject of further study.

Moreover, integrating the rippling work of Section 9.4 and *wpin* (*win*) with topology and data-flow analysis should provide a very good framework for automation. Here, topology and data-flow analysis could be used to identify the type of invariant. Based on this identification the correct technique could be applied, e.g. *wpin* in the case of a (D2) invariant, rippling in the case of a (D3) invariant, and *win* in the case of a (D4) invariant.

Theorem proving and model checking have been separately discussed. Addressing this integration seems particularly important for Hierarchical Hume. For example, model checking "nested properties", with a theorem prover in charge of the overall

proof. This may help to avoid the state-space explosion problem, since it should be sufficient to enumerate the full state space of the nesting box. A direct integration using TLC as an oracle has the potential of introducing true-negatives, since both TLC and the translations between Isabelle/TLA and TLA$^+$ have to be trusted. A more mathematical rigourous interface, as for example used for the μ-calculus and HOL in [15], requires a much more substantial work load.

The first two case-studies in Chapter 7 verified transformations. The proofs relied on the "result buffer property": the (internal) output wires of the resulting box, must with the same input value, have the same values as computed by the expression of the source box (see Section 6.6). If this property holds, then minimal user interaction is required when applying the transformation tactic. Proof planning has previously been used for transformation verification [136, 46], which may also be applicable here. In particular, in finding and verifying the "result buffer property", by also integrating the weakest precondition and loop invariant work discussed above.

10.3.4 The box calculus

The work on the box calculus at the lowest, least expressive HW-Hume level gives confidence in the calculus and allows focus on the intricate properties of the coordination layer, which are the same for all Hume levels. Extending the calculus beyond HW-Hume will mainly require an extension of the purely functional transformation rules, together with data refinement. A next step here is to identify a sufficient set of rules which is adequate for the classes of transformations between and within levels. This will enable support to tackle the problems with respect to costing, which is the main motivation behind Hume transformations. Moreover, to support more flexibility it may be necessary to incorporate rules which are not behaviour preserving on their own, but which can be combined into "correct" rules/strategies.

Besides this "ability to cost programs" motivation, the calculus can be used for optimisation. This is particularly interesting, and non-trivial, in Hume since the different layers will implement the same algorithm completely differently. This can be achieved by augmenting the rule with the resource cost model, i.e. the change of resource properties, and only apply resource reducing rules and strategies. This will provide a way of implementing the *costing by construction* principle suggested for Hume [175]. This principle is refinement-based and cost driven. Here, the coordination structure is first defined, followed by a phase implementing the body of the boxes, and finally, a validation phase. In the validation phase, program refinement (transformation) is driven by the costing, and this phase could be implemented by the box calculus augmented by the cost model. Moreover, incorporating preservation of liveness with the rules, is also

a matter for future research.

The calculus will be hard to deploy without proper tool support. Firstly, its logical details must be mechanised. A direct mechanisation of the calculus as described here, requires a deep mechanisation of Hierarchical Hume in Isabelle/TLA. Notions like **Time_Dependency** (Figure 8.2 on page 167) must then be formalised, and the rules and strategies derived.

However, a first and more lightweight approach, is to indirectly use the calculus in the current embedding by representing rules and strategies as tactics or proof plans [46, 136]. This approach will still be ad-hoc, since the transformations are conducted first, and then the verification is attempted. However, a rule application should have a similar structure regardless of the program, thus the tactic or proof plan should be able to capture this.

The calculus is defined for the Hume (source code) level. Thus, the user should not see the underlying logical details. Instead, determining the rules and strategies to apply should either be automatic, or by selecting a rule or strategy from a menu in a Hume IDE, like in the HaRe system [183] for refactoring Haskell programs. Finally, in a failed proof, the failure needs to be translated into Hume source code.

With the exception of the box calculus, the verification process in this thesis has been ad-hoc: a transformation, or a program with specified properties, is first developed and then verification is attempted. The box calculus is a first step towards guiding the development by proofs. However, the formal methods are still applied late in the development process. Following the B-method, it would be interesting to specify a Hume program in TLA, refine this into a suitable subset, and synthesise boxes and wires. Such an approach could enable integration of the implementation and modelling phases, where some boxes are only specified in TLA, whilst others are implemented. Moreover, since Hume is encoded in TLA, and the reasoning is within TLA, hardware components can be specified in TLA, which enables hardware/software co-design, one of the targets of the HW-Hume level [91]. There is also a lower, intermediate *Hume Abstract Machine* (HAM) level, which the Hume code is compiled to, and to which some resource analyses are applied. TLA can be applied to HAM, and be used to verify the Hume to HAM translation.

10.3.5 Towards a Hume verification environment

Once integration with the expression layer is more mature, with a specification language and translator, a significant step towards the long term goal of building a verification environment for Hume will have been achieved. Here, the vision is to provide certified software, where a program is accompanied with a correctness proof, similarly to

McKinna's *deliverables* [139] and Necula's *proof carrying code* [154].

Such an environment should also incorporate the costing tools. Moreover, a graphical user interface, which can simulate program execution would be highly desirable particularly, to reproduce counter examples from failed verification tasks. More semi-formal tools, such as testing, for non-critical parts of programs are also worth exploring.

A final question is how this work with Hume and TLA can be extended beyond the Hume context. The TLA mechanisation can be used for any TLA embedded system, and the none Hume-specific tactics, such as invariant strengthening and TLA simplification can also be used in this context. Moreover, the mechanisation of sequences can be used with other logics. The remaining tactics are very much geared towards Hume, and are unlikely to be directly re-used in other contexts. However, some parts, such as the automatic unlifting from TLA into HOL, can be separated out, and used to help verify generic TLA invariants and refinements. Proofs of such generic TLA invariants, as explained in a Disc Paxos paper [73] for example, are based upon building up a strong enough invariant from smaller ones. Due to its inductive nature, the rippling approach, described in Section 9.4, should be very much applicable for such proofs in a backward fashion. Moreover, rippling has previously been used to find complex witnesses [141], and may thus be used to find refinement mappings.

10.4 Concluding remarks

The results of using TLA for Hume coordination verification are very encouraging. The formalism appears to be ideal for Hume's design, and through the use of proof tactics has shown that automation is possible. Moreover, by exploiting the lifted nature of TLA, integration with other reasoning techniques was very direct. Thus, it should be ideal for generic coordination languages, like HTL [77], which are independent of the computation inside the components. However, it is assumed that the communication is state-based.

The work with transformations is also very encouraging. Following the motivation behind a Hume transformation, the source of the transformation is always one box. Thus, by the introduction of Hierarchical Hume, the simple prototype tactics were able to achieve a substantial degree of automation. The advantages of introducing Hierarchical Hume have also been shown, and such structuring is important, particularly for scalability.

To conclude, the use of TLA to reason at the programming level, is very promising, particularly for coordination languages. Moreover, the results with Hierarchical Hume are very encouraging, and should be incorporated into all Hume tools, where the goal

is to deploy Hume on industrial-sized programs.

Appendix A

Mechanised theorems in Isabelle/HOL

Here, the lemmas and theorems which have been mechanically proved in Isabelle/HOL are listed.

A.1 The TLA mechanisation

A.1.1 Intensional

theorem `intI`: **assumes:** $\bigwedge$ w. w $\vDash$ A **shows:** $\vdash$ A
theorem `intD`: **assumes:** $\vdash$ A **shows:** $\bigwedge$ w. w $\vDash$ A
theorem `inteq_reflection`: **assumes:** $\vdash$ x=y **shows:** x $\equiv$ y
theorem `int_iffI`: **assumes:** $\vdash$ F $\longrightarrow$ G **and** $\vdash$ G $\longrightarrow$ F **shows:** $\vdash$ F = G
theorem `int_iffD1`: **assumes:** $\vdash$ F = G **shows:** $\vdash$ F $\longrightarrow$ G
theorem `int_iffD2`: **assumes:** $\vdash$ F = G **shows:** $\vdash$ G $\longrightarrow$ F
theorem `imp_tr`: **assumes:** $\vdash$ A $\longrightarrow$ B **and** $\vdash$ B $\longrightarrow$ C **shows:** $\vdash$ A $\longrightarrow$ C

A.1.2 Sequences

lemma `suffix_first`: first (s $|_s$ n) = s n
lemma `suffix_plus`: s $|_s$ n $|_s$ m = s $|_s$ (m + n)
lemma `suffix_commute`: ((s $|_s$ n) $|_s$ m) = ((s $|_s$ m) $|_s$ n)
lemma `suffix_zero`: s $|_s$ 0 = s
lemma `suffix_tail`: s $|_s$ 1 = tail s
lemma `tail_suffix_suc`: s $|_s$ (Suc n) = tail (s $|_s$ n)
lemma `seq_app_first_tail`: (first s) ## (tail s) = s
lemma `seq_app_tail`: tail (x ## s) = s
lemma `seq_app_greather_than_zero`: $\forall$ n > 0. (s ## σ) n = σ (n-1)

lemma finite_or_infinite: $\forall$ s. fin s $\vee$ inf s
lemma not_finite_is_infinite: $(\neg$ fin s$) =$ inf s
lemma not_infinite_is_finite: $(\neg$ inf s$) =$ fin s
lemma empty_is_finite: **assumes:** emptyseq s **shows:** fin s
lemma empty_suffix_is_empty:
assumes: emptyseq s **shows:** $\forall$ n. emptyseq (s $|_s$ n)
lemma empty_suffix_exteq:
assumes: emptyseq s **shows:** $\forall$ m. (s $|_s$ n) m $=$ s m
lemma seq_empty_or_notempty: emptyseq s $\neq$ notemptyseq s
lemma suc_empty:
assumes: emptyseq (s $|_s$ m) **shows:** emptyseq (s $|_s$ (Suc m))
lemma fin_stut_after_last:
assumes: fin s **shows:** $\forall$ j $\geq$ (last s). s j $=$ s (last s)

Stuttering invariance

lemma seq_empty_is_nonstut: **assumes:** emptyseq s **shows:** nonstutseq s
lemma notempty_exist_nonstut:
assumes: emptyseq (s $|_s$ m) **shows:** $\exists$ i. s i $\neq$ s m $\wedge$ i $>$ m
lemma nextnat_le_unch: **assumes:** n $<$ nextnat s **shows:** s n $=$ s 0
lemma stutnempty: **assumes:** $\neg$ stutstep s n **shows:** $\neg$ emptyseq (s $|_s$ n)
lemma notstutstep_nexnat1:
assumes: $\neg$ stutstep s n **shows:** nextnat (s $|_s$ n) $=$ 1
lemma stutstep_notempty2: **assumes:** notemptyseq (s $|_s$ n) **and** stutstep s n
shows: notemptyseq (s $|_s$ (Suc n))
lemma stutstep_notempty_sucnextnat:
assumes: $\neg$ emptyseq (s $|_s$ n) **and** stutstep s n
shows: (nextnat (s $|_s$ n)) $=$ Suc (nextnat (s $|_s$ (Suc n)))
lemma nextn_empty_neq: **assumes:** $\neg$ emptyseq s **shows:** s (nextnat s) $\neq$ s 0
lemma nextn_empty_gzero: **assumes:** $\neg$ emptyseq s **shows:** nextnat s $>$ 0
lemma empty_nextsuffix: **assumes:** emptyseq s **shows:** nextsuffix s $=$ s
lemma empty_nextsuff_id: **assumes:** emptyseq s **shows:** nextsuffix s $=$ id s
lemma notstutstep_nextsuffix1:
assumes: $\neg$ stutstep s n **shows:** nextsuffix (s $|_s$ n) $=$ s $|_s$ (Suc n)
lemma next_first: next 1 $=$ nextsuffix
lemma next_zero: next 0 s $=$ s
lemma next_suc_suffix: next (Suc n) s $=$ nextsuffix (next n s)
lemma next_suffix_com: nextsuffix (next n s) $=$ (next n (nextsuffix s))

lemma next_plus: next (m+n) s = next m (next n s)
lemma next_empty: **assumes:** emptyseq s **shows:** next n s = s
lemma notempty_nextnzero:
assumes: ¬ emptyseq s **shows:** (next (Suc 0) s) 0 ≠ s 0
lemma next_ex_id: ∃ i. s i = (next m s) 0
lemma emptyseq_collapse_eq: **assumes:** emptyseq s **shows:** ♮ s = s
lemma empty_collapse_empty: **assumes:** emptyseq s **shows:** emptyseq (♮ s)
lemma collapse_empty_empty: **assumes:** emptyseq (♮ s) **shows:** emptyseq s
lemma notempty_collapse_notempty:
assumes: notemptyseq s **shows:** notemptyseq (♮ s)
lemma collapse_notempty_notempty:
assumes: notemptyseq (♮ s) **shows:** notemptyseq s
lemma seqsim_refl: s ≈ s
lemma seqsim_sym: **assumes:** s ≈ t **shows:** t ≈ s
lemma seqeq_imp_sim: **assumes:** s = t **shows:** s ≈ t
lemma seqsim_trans: **assumes:** s ≈ t **and** t ≈ z **shows:** s ≈ z
lemma sim_first: **assumes:** s ≈ t **shows:** first s = first t
lemma seqsim_empty2: **assumes:** s ≈ t **and** emptyseq s **shows:** emptyseq t
lemma seq_empty_eq: **assumes:** s 0 = t 0 **and** emptyseq s **and** emptyseq t
shows: s = t
lemma coleq_seqsim: (♮ s = ♮ t) = (s ≈ t)
lemma seqsim_notempty_notempty:
assumes: s ≈ t **and** notemptyseq s **shows:** notemptyseq t
lemma seqsim_notstutstep:
assumes: ¬ (stutstep s n) **shows:** (s |$_s$ (Suc n)) ≈ nextsuffix (s |$_s$ n)k"
lemma stut_nextsuf_suc: **assumes:** stutstep s n
shows: nextsuffix (s |$_s$ n) = nextsuffix (s |$_s$ (Suc n))
lemma seqsim_suffix_seqsim: **assumes:** s ≈ t
shows: nextsuffix s ≈ nextsuffix t
lemma seqsim_stutstep: **assumes:** stutstep s n
shows: (s |$_s$ (Suc n)) ≈ (s |$_s$ n)
lemma add_feqstut: **assumes:** s = first t **shows:** stutstep (s ## t) 0
lemma add_feqsim: **assumes:** s = first t **shows:** (s ## t) ≈ t
lemma add_first_stut: **assumes:** first s = second s **shows:** s ≈ tail s
theorem app_similar: **assumes:** s ≈ t **shows:** (x ## s) ≈ (x ## t)
lemma simstep_disj1: **assumes:** s ≈ t **shows:** ∀ n. ∃ m. ((s |$_s$ n) ≈ (t |$_s$ m))
lemma nextnat_le_seqsim: n < nextnat s ⟶ s ≈ (s |$_s$ n)

lemma `seqsim_prev_nextnat`: $s \approx s\,|_s\,((\text{nextnat } s)-1)$
theorem `sim_step (first attempt)`: **assumes:** $s \approx t$
shows: $\forall n.\ \exists m.\ (s\,|_s n \approx t\,|_s m) \wedge (\ s\,|_s(\text{Suc } n) \approx t\,|_s(\text{Suc } m))$

A.1.3 Semantics

theorem `ev1`: $(w \vDash \Diamond F) = (\exists n.\ (w\,|_s\ n) \vDash F)$
theorem `aact1`: $(w \vDash \Diamond \langle\ P\ \rangle_v) = (\exists i.\ ((w\,|_s\ i) \vDash P) \wedge ((w\,|_s\ i) \vDash v\$ \neq \$v))$

Stuttering invariance

theorem `stutinv_nstutinv1`: **assumes:** STUTINV F **shows:** NSTUTINV F
theorem `stut_before`: STUTINV $F
theorem `stut_always`: **assumes:** STUTINV F **shows:** STUTINV $\Box$F
theorem `stut_action`: **assumes:** NSTUTINV P **shows:** STUTINV $\Box[P]_v$
theorem `stut_const`: STUTINV #c
theorem `stut_fun1`: **assumes:** STUTINV x **shows:** STUTINV (f <x>)
theorem `stut_fun2`:
assumes: STUTINV x **and** STUTINV y **shows:** STUTINV (f <x,y>)
theorem `stut_fun3`:
assumes: STUTINV x **and** STUTINV y **and** STUTINV z
shows: STUTINV (f <x,y,z>)
theorem `stut_and`:
assumes: STUTINV F **and** STUTINV G **shows:** STUTINV (F $\wedge$ G)
theorem `stut_or`:
assumes: STUTINV F **and** STUTINV G **shows:** STUTINV (F $\vee$ G)
theorem `stut_imp`:
assumes: STUTINV F **and** STUTINV G **shows:** STUTINV (F $\longrightarrow$ G)
theorem `stut_eq`:
assumes: STUTINV F **and** STUTINV G **shows:** STUTINV (F = G)
theorem `stut_neq`:
assumes: STUTINV F **and** STUTINV G **shows:** STUTINV (F $\neq$ G)
theorem `stut_not`: **assumes:** STUTINV F **shows:** STUTINV ($\neg$ F)
theorem `stut_exists`: **assumes:** STUTINV F **shows:** STUTINV ($\exists$ x. F)
theorem `stut_forall`: **assumes:** STUTINV F **shows:** STUTINV ($\forall$ x. F)
theorem `nstut_nexts`: **assumes:** STUTINV F **shows:** NSTUTINV $\circ$F
theorem `nstut_const`: NSTUTINV #c
theorem `nstut_fun1`: **assumes:** NSTUTINV x **shows:** STUTINV (f <x>)

theorem nstut_fun2:
assumes: NSTUTINV x **and** NSTUTINV y **shows:** NSTUTINV (f <x,y>)
theorem nstut_fun3:
assumes: NSTUTINV x **and** NSTUTINV y **and** NSTUTINV z
shows: NSTUTINV (f <x,y,z>)
theorem nstut_and:
assumes: NSTUTINV F **and** NSTUTINV G **shows:** NSTUTINV (F ∧ G)
theorem nstut_or:
assumes: NSTUTINV F **and** NSTUTINV G **shows:** NSTUTINV (F ∨ G)
theorem nstut_imp:
assumes: NSTUTINV F **and** NSTUTINV G **shows:** NSTUTINV (F ⟶ G)
theorem nstut_eq:
assumes: NSTUTINV F **and** NSTUTINV G **shows:** NSTUTINV (F = G)
theorem nstut_neq:
assumes: NSTUTINV F **and** NSTUTINV G **shows:** NSTUTINV (F ≠ G)
theorem nstut_not: **assumes:** NSTUTINV F **shows:** NSTUTINV (¬ F)
theorem nstut_exists: **assumes:** NSTUTINV F **shows:** NSTUTINV (∃ x. F)
theorem nstut_forall: **assumes:** NSTUTINV F **shows:** NSTUTINV (∀ x. F)
theorem nstut_after: NSTUTINV F
theorem nstut_unch: NSTUTINV (Unchanged v)
theorem nstut_actrans: **assumes:** NSTUTINV P **shows:** NSTUTINV [P]_v
theorem stut_ev: **assumes:** STUTINV F **shows:** STUTINV ◇F
theorem stut_aa: **assumes:** NSTUTINV P **shows:** STUTINV ◇⟨P⟩_v
theorem nstut_aa: **assumes:** NSTUTINV P **shows:** NSTUTINV ⟨P⟩_v

A.1.4 PreFormulas

theorem prefI: **assumes:** ⋀ w. w ⊨ A **shows:** ⊢∼A
theorem prefD: **assumes:** ⊢∼A **shows:** ⋀ w. w ⊨ A
theorem prefeq_reflection: **assumes:** ⊢∼x=y **shows:** x ≡ y
lemma pre_id_unch: **assumes:** stutinv F **shows:** ⊢∼ F ∧ Unchanged id ⟶ ∘F
lemma pre_ex_unch:
assumes: stutinv F **shows:** ∃ v::('a ⇒ 'a). ⊢∼ F ∧ Unchanged v ⟶ ∘F
lemma unch_pair: ⊢∼ Unchanged (x,y) = (Unchanged x ∧ Unchanged y)
lemma angle_actrans_sem: ⊢∼ (⟨F⟩_v) = (F ∧ ((v$) ≠ $v))
lemma after_const: ⊢∼ (#c)$ = #c
lemma after_fun1: ⊢∼ f<x>$ = f <x$>

lemma `after_fun2:` $\vdash$ f<x,y>$ = f <x$,y$>
lemma `after_fun3:` $\vdash$ f<x,y,z>$ = f <x$,y$,z$>
lemma `after_fun4:` $\vdash$ f<x,y,z,zz>$ = f <x$,y$,z$,zz$>
lemma `after_forall:` $\vdash$ (∀ x. P x)$ = (∀ x. (P x)$)
lemma `after_exists:` $\vdash$ (∃ x. P x)$ = (∃ x. (P x)$)
lemma `after_exists1:` $\vdash$ (∃! x. P x)$ = (∃! x. (P x)$)
lemma `after_pair:` $\vdash$ (X,Y)$ = (X$,Y$)
lemma `after_le:` $\vdash$ (x < y)$ = (x$ < y$)
lemma `after_le_eq:` $\vdash$ (x ≤ y)$ = (x$ ≤ y$)
lemma `after_mem:` $\vdash$ (x ∈ X)$ = (x$ ∈ X$)
lemma `after_notmem:` $\vdash$ (x ∉ X)$ = (x$ ∉ X$)
lemma `after_if:` $\vdash$ (if T then A else B)$ = (if T$ then A$ else B$)
lemma `after_plus:` $\vdash$ (x + y)$ = x$ + y$
lemma `after_minus:` $\vdash$ (x - y)$ = x$ - y$
lemma `after_times:` $\vdash$ (x * y)$ = x$ * y$
lemma `after_div:` $\vdash$ (x div y)$ = x$ div y$
lemma `after_mod:` $\vdash$ (x mod y)$ = x$ mod y$
lemma `after_finset:` $\vdash$ {X}$ = {X$}
lemma `after_cons:` $\vdash$ (x#xs)$ = (x$#xs$)
lemma `after_app:` $\vdash$ (xs@ys)$ = (xs$@ys$)
lemma `after_list:` $\vdash$ [L]$ = [L$]
lemma `after_if:` $\vdash$ (if A then B else C)$ = (if A$ then B$ else C$)
lemma `before_const:` $\vdash$ $(#c) = #c
lemma `before_fun1:` $\vdash$ $(f<x>) = f <$x>
lemma `before_fun2:` $\vdash$ $(f<x,y>) = f <$x,$y>
lemma `before_fun3:` $\vdash$ $(f<x,y,z>) = f <$x,$y,$z>
lemma `before_fun4:` $\vdash$ $(f<x,y,z,zz>) = f <$x,$y,$z,$zz>
lemma `before_forall:` $\vdash$ $(∀ x. P x) = (∀ x. $(P x))
lemma `before_exists:` $\vdash$ $(∃ x. P x) = (∃ x. $(P x))
lemma `before_exists1:` $\vdash$ $(∃! x. P x) = (∃! x. $(P x))
lemma `before_pair:` $\vdash$ $(X,Y) = ($X,$Y)
lemma `before_le:` $\vdash$ $(x < y) = ($x < $y)
lemma `before_le_eq:` $\vdash$ $(x ≤ y) = ($x ≤ $y)
lemma `before_mem:` $\vdash$ $(x ∈ X) = ($x ∈ $X)
lemma `before_notmem:` $\vdash$ $(x ∉ X) = ($x ∉ $X)
lemma `before_if:` $\vdash$ $(if T then A else B) = (if $T then $A else $B)
lemma `before_plus:` $\vdash$ $(x + y) = ($x) + ($y)

lemma `before_minus`: $\vdash \$(x - y) = \$x - \$y$
lemma `before_times`: $\vdash \$(x * y) = \$x * \$y$
lemma `before_div`: $\vdash \$(x \text{ div } y) = \$x \text{ div } \$y$
lemma `before_mod`: $\vdash \$(x \text{ mod } y) = \$x \text{ mod } \$y$
lemma `before_finset`: $\vdash \$\{X\} = \{\$X\}$
lemma `before_cons`: $\vdash \$(x\#xs) = (\$x\#\$xs)$
lemma `before_app`: $\vdash \$(xs@ys) = (\$xs@\$ys)$
lemma `before_list`: $\vdash \$[L] = [\$L]$
lemma `before_if`: $\vdash \$(\text{if A then B else C}) = (\text{if } \$A \text{ then } \$B \text{ else } \$C)$

A.1.5 Rules

The basic axioms

theorem `fmp`: **assumes:** $\vdash F$ **and** $\vdash F \longrightarrow G$ **shows:** $\vdash G$
theorem `pmp`: **assumes:** $\mathrel{|\!\sim} F$ **and** $\mathrel{|\!\sim} F \longrightarrow G$ **shows:** $\mathrel{|\!\sim} G$
theorem `sq`: **assumes:** $\mathrel{|\!\sim} P$ **shows:** $\vdash \Box[P]\_v$
theorem `pre`: **assumes:** $\vdash F$ **shows:** $\mathrel{|\!\sim} F$
theorem `nex`: **assumes:** $\vdash F$ **shows:** $\mathrel{|\!\sim} \circ F$
theorem `ax0`: $\vdash \#\text{True}$
theorem `ax1`: $\vdash \Box F \longrightarrow F$
theorem `ax2`: $\vdash \Box F \longrightarrow \Box[\Box F]\_v$
theorem `ax3`: **assumes:** $\mathrel{|\!\sim} F \wedge \text{Unchanged } v \longrightarrow \circ F$
shows: $\vdash \Box[F \longrightarrow \circ F]\_v \longrightarrow (F \longrightarrow \Box F)$
theorem `ax4`: $\vdash \Box[P \longrightarrow Q]\_v \longrightarrow (\Box[P]\_v \longrightarrow \Box[Q]\_v)$
theorem `ax5`: $\vdash \Box[v\$ \neq \$v]\_v$
theorem `pax0`: $\mathrel{|\!\sim} \#\text{True}$
theorem `pax1`: $\mathrel{|\!\sim} (\circ\neg F) = (\neg\circ F)$
theorem `pax2`: $\mathrel{|\!\sim} \circ(F \longrightarrow G) \longrightarrow (\circ F \longrightarrow \circ G)$
theorem `pax3`: $\mathrel{|\!\sim} \Box F \longrightarrow \circ\Box F$
theorem `pax4`: $\mathrel{|\!\sim} (\Box[P]\_v) = ([P]\_v \wedge \circ\Box[P]\_v)$
theorem `pax5`: $\mathrel{|\!\sim} \circ\Box F \longrightarrow \Box[\circ F]\_v$

Derived lemmas from [144]

lemma `allT`: $\vdash (\forall x.\ \Box(F\ x)) = (\Box(\forall x.\ F\ x))$
lemma `allActT`: $\vdash (\forall x.\ \Box[(F\ x)]\_v) = (\Box[(\forall x.\ F\ x)]\_v)$

lemma `alw`: **assumes:** $\vdash F$ **shows:** $\vdash \Box F$
lemma `alw2`: **assumes:** $\vdash F$ **and** $\vdash\!\!\sim F \wedge \mathrm{Unchanged}\ v \longrightarrow \circ F$ **shows:** $\vdash \Box F$
lemma `alw3`: **assumes:** $\vdash F$ **and** stutinv F **shows:** $\vdash \Box F$
lemma `T1`: $\vdash (\Box F) = (\Box\Box F)$
lemma `T2`: $\vdash (\Box [P]_v) = (\Box\Box [P]_v)$
lemma `T3`: $\vdash (\Box [[P]_v]_v) = (\Box [P]_v)$
lemma `T4`: $\vdash \Box [P]_v \longrightarrow \Box [[P]_v]_w$
lemma `T5`: $\vdash \Box [[P]_w]_v \longrightarrow \Box [[P]_v]_w$
lemma `T6`: $\vdash \Box F \longrightarrow \Box [\circ F]_v$
lemma `T7`: **assumes:** $\vdash\!\!\sim F \wedge \mathrm{Unchanged}\ v \longrightarrow \circ F$
shows: $\vdash\!\!\sim (\Box F) = (F \wedge \circ\Box F)$
lemma `T8`: $\vdash\!\!\sim (\circ(F \wedge G)) = (\circ F \wedge \circ G)$
lemma `H1`: **assumes:** $\vdash \Box [P]_v$ **and** $\vdash \Box [P \longrightarrow Q]_v$ **shows:** $\vdash \Box [Q]_v$
lemma `H2`: **assumes:** $\vdash F$ **shows:** $\vdash \Box [F]_v$
lemma `H3`: **assumes:** $\vdash \Box [P \longrightarrow Q]_v$ **and** $\vdash \Box [Q \longrightarrow R]_v$
shows: $\vdash \Box [P \longrightarrow R]_v$
lemma `H4`: $\vdash \Box [[P]_v \longrightarrow P]_v$
lemma `H5`: $\vdash \Box [\Box F \longrightarrow \circ\Box F]_v$

Various derived lemmas

lemma `P1`: $\vdash\!\!\sim \Box F \longrightarrow \circ F$
lemma `P2`: $\vdash\!\!\sim \Box F \longrightarrow F \wedge \circ F$
lemma `P3`: $\vdash\!\!\sim \Box F \longrightarrow F \longrightarrow \circ F$
lemma `P4`: $\vdash \Box F \longrightarrow \Box [F]_v$
lemma `P5`: $\vdash \Box [P]_v \longrightarrow \Box [\Box [P]_v]_w$
lemma `M0`: $\vdash \Box F \longrightarrow \Box [(F \longrightarrow \circ F)]_v$
lemma `M1`: $\vdash \Box F \longrightarrow \Box [(F \wedge \circ F)]_v$
lemma `M2`: **assumes:** $\vdash F \longrightarrow G$ **shows:** $\vdash \Box [F]_v \longrightarrow \Box [G]_v$
lemma `M3`: **assumes:** $\vdash F$ **shows:** shows $\vdash \Box [\circ F]_v$
lemma `M4`: $\vdash \Box [(\circ(F \wedge G)) = (\circ F \wedge \circ G)]_v$
lemma `M5`: $\vdash \Box [(\Box [P]_v \longrightarrow \circ\Box [P]_v)]_w$
lemma `M6`: $\vdash \Box [F \wedge G]_v \longrightarrow \Box [F]_v \wedge \Box [G]_v$
lemma `M7`: $\vdash \Box [F]_v \wedge \Box [G]_v \longrightarrow \Box [F \wedge G]_v$
lemma `M8`: $\vdash (\Box [F]_v \wedge \Box [G]_v) = (\Box [F \wedge G]_v)$
lemma `M9`: $\vdash\!\!\sim \Box F \longrightarrow F \wedge \circ\Box F$
lemma `M10`: **assumes:** $\vdash\!\!\sim F \wedge \mathrm{Unchanged}\ v \longrightarrow \circ F$ **shows:** $\vdash\!\!\sim F \wedge \circ\Box F \longrightarrow \Box F$
lemma `STL2`: $\vdash \Box F \longrightarrow F$

lemma STL3: $\vdash (\Box F) = (\Box\Box F)$

lemma STL4: **assumes:** $\vdash F \longrightarrow G$ **shows:** $\vdash \Box F \longrightarrow \Box G$

lemma STL4_2: **assumes:** $\vdash F \longrightarrow G$ **and** $\vdash\!\!\sim G \wedge \mathrm{Unchanged}\ v \longrightarrow \circ G$
shows: $\vdash \Box F \longrightarrow \Box G$

lemma STL4_3: **assumes:** $\vdash F \longrightarrow G$ **and** STUTINV G **shows:** $\vdash \Box F \longrightarrow \Box G$

lemma STL5: $\vdash (\Box(F \wedge G)) = (\Box F \wedge \Box G)$

lemma STL5_2:
assumes: $\vdash\!\!\sim F \wedge \mathrm{Unchanged}\ v1 \longrightarrow \circ F$ **and** $\vdash\!\!\sim G \wedge \mathrm{Unchanged}\ v2 \longrightarrow \circ G$
and $\vdash\!\!\sim (F \wedge G) \wedge \mathrm{Unchanged}\ v3 \longrightarrow \circ(F \wedge G)$
shows: $\vdash (\Box(F \wedge G)) = (\Box F \wedge \Box G)$

lemma STL5_21:
assumes: stutinv F **and** stutinv G **shows:** $\vdash (\Box(F \wedge G)) = (\Box F \wedge \Box G)$

lemma MM7: **assumes:** $s \models \Box F$ **and** $s \models \Box G$ **and** $s \models \Box(F \wedge G) \longrightarrow$ PROP R
shows: PROP R

lemma linalw: **assumes:** $a \leq b$ **and** $(w \mid_s a) \models \Box A$ **shows:** $(w \mid_s b) \models \Box A$

lemma STL6: $\vdash ((\Diamond\Box F) \wedge (\Diamond\Box G)) = (\Diamond\Box(F \wedge G))$

lemma MM1: **assumes:** $\vdash F = G$ **shows:** $\vdash (\Box F) = (\Box G)$

lemma MM2: $\vdash \Box A \wedge \Box(B \longrightarrow C) \longrightarrow \Box(A \wedge B \longrightarrow C)$

lemma MM3: $\vdash \Box(\neg A) \longrightarrow \Box(A \wedge B \longrightarrow C)$

lemma MM4: $\vdash (\#F) = (\Box(\#F))$

lemma MM4b: $\vdash (\neg\#F) = (\Box(\neg\#F))$

lemma MM5: $\vdash \Box F \vee \Box G \longrightarrow \Box(F \vee G)$

lemma MM6: $\vdash \Box F \vee \Box G \longrightarrow \Box(\Box F \vee \Box G)$

lemma MM7: $\vdash \Box(\Box F \longrightarrow G) \wedge \Box F \longrightarrow \Box G$

lemma MM8: **assumes:** $\vdash\!\!\sim F \longrightarrow G$ **shows:** $\vdash \Box[F]_v \longrightarrow \Box[G]_v$

lemma MM9: **assumes:** $\vdash F = G$ **shows:** $\vdash (\Box[F]_v) = (\Box[G]_v)$

lemma MM10: **assumes:** $\vdash\!\!\sim F = G$ **shows:** $\vdash (\Box[F]_v) = (\Box[G]_v)$

lemma MM11: $\vdash \Box[\neg(P \wedge Q)]_v \longrightarrow \Box[P]_v \longrightarrow \Box[(P \wedge \neg Q)]_v$

lemma MM12: $\vdash (\Box[\Box[P]_v]_v) = (\Box[P]_v)$

lemma E1: $\vdash (\Diamond(F \vee G)) = (\Diamond F \vee \Diamond G)$

lemma E2: $\vdash (\Box F) = (\neg\Diamond\neg F)$

lemma E3: $\vdash F \longrightarrow \Diamond F$

lemma E4: $\vdash \Box F \longrightarrow \Diamond F$

lemma E5: $\vdash \Box F \longrightarrow \Box\Diamond F$

lemma E6: $\vdash \Box F \longrightarrow \Diamond\Box F$

lemma E7: **assumes:** $\vdash\!\!\sim \neg F \wedge \mathrm{Unchanged}\ v \longrightarrow \circ\neg F$
shows: $\vdash\!\!\sim \Diamond F \longrightarrow F \vee \circ\Diamond F$

lemma E8: $\vdash \Diamond(F \longrightarrow G) \longrightarrow \Box F \longrightarrow \Diamond G$
lemma E9: $\vdash \Box(F \longrightarrow G) \longrightarrow \Diamond F \longrightarrow \Diamond G$
lemma E10: $\vdash (\Diamond F) = (\Diamond\Diamond F)$
lemma E11: $\vdash \Box\Diamond F \longrightarrow \Diamond F$
lemma E12: $\vdash (\neg\Diamond\Box F) = (\Box\Diamond\neg F)$
lemma E13: $\vdash (\Diamond\Box\neg F) = (\neg\Box\Diamond F)$
lemma E15: $\vdash (\#F) = (\Diamond(\#F))$
lemma E15b: $\vdash (\neg\#F) = (\Diamond(\neg\#F))$
lemma E16: $\vdash \Diamond\Box F \longrightarrow \Diamond F$
lemma E17: $\vdash \Box\Diamond\Box F \longrightarrow \Box\Diamond F$
lemma STL6_act: $\vdash ((\Diamond\Box[F]_v) \wedge (\Diamond\Box[G]_w)) = (\Diamond(\Box[F]_v \wedge \Box[G]_w))$
lemma STL6_E1: $\vdash \Box F \wedge \Diamond G \longrightarrow \Diamond(\Box F \wedge G)$
lemma STL6_E2: $\vdash \Box F \wedge \Diamond G \longrightarrow \Diamond(F \wedge G)$
lemma STL6_E4: **assumes:** $s \vDash \Box A$ **and** $s \vDash \Diamond F$ **and** $\vdash \Box A \wedge F \longrightarrow G$
shows: $s \vDash \Diamond G$
lemma E18: $\vdash \Box\Diamond\Box F \longrightarrow \Diamond\Box F$
lemma E19: $\vdash \Diamond\Box F \longrightarrow \Box\Diamond\Box F$
lemma E20: $\vdash \Diamond\Box F \longrightarrow \Box\Diamond F$
lemma E21: $\vdash (\Box\Diamond\Box F) = (\Diamond\Box F)$
lemma E22: **assumes:** $\vdash F = G$ **shows:** $\vdash (\Diamond F) = (\Diamond G)$
lemma E23: $\vdash\!\!\!\sim \circ F \longrightarrow \Diamond F$
lemma E24: $\vdash \Diamond\Box Q \longrightarrow \Box[\Diamond Q]_v$
lemma E25: $\vdash \Diamond\langle A\rangle_v \longrightarrow \Diamond A$
lemma E26: $\vdash \Box\Diamond\langle A\rangle_v \longrightarrow \Box\Diamond A$
lemma allBox: $\vdash \Box(\forall x.\ F\ x)) = (\forall x.\ s \vdash \Box(F\ x))$
lemma E27: $\vdash (\Diamond\Box\Diamond F) = (\Box\Diamond F)$
lemma E28: $\vdash \Diamond\Box F \wedge \Box\Diamond G \longrightarrow \Box\Diamond(F \wedge G)$
lemma E29: $\vdash\!\!\!\sim \circ\Diamond F \longrightarrow \Diamond F$
lemma allActBox: $s \vDash \Box[(\forall x.\ F\ x)]_v) = (\forall x.\ s \vDash \Box[(F\ x)]_v)$
lemma exEE: $\vdash (\exists x.\ \Diamond(F\ x)) = (\Diamond(\exists x.\ F\ x))$
lemma exActE: $\vdash (\exists x.\ \Diamond\langle F\ x\rangle_v) = (\Diamond\langle(\exists x.\ F\ x)\rangle_v)$
lemma LT1: $\vdash F \rightsquigarrow F$
lemma LT2: **assumes:** $\vdash F \longrightarrow G$ **shows:** $\vdash F \longrightarrow \Diamond G$
lemma LT3: **assumes:** $\vdash F \longrightarrow G$ **shows:** $\vdash F \rightsquigarrow G$
lemma LT4: $\vdash F \wedge (F \rightsquigarrow G) \longrightarrow \Diamond G$
lemma LT5: $\vdash \Box(F \longrightarrow \Diamond G) \longrightarrow \Diamond F \longrightarrow \Diamond G$
lemma LT6: $\vdash \Diamond F \wedge (F \rightsquigarrow G) \longrightarrow \Diamond G$

lemma LT7: $\vdash \Box\Diamond F \wedge (F \rightsquigarrow G) \longrightarrow \Box\Diamond G$

lemma LT8: $\vdash \Box\Diamond G \longrightarrow (F \rightsquigarrow G)$

lemma LT11: $\vdash (F \rightsquigarrow G) \longrightarrow (F \rightsquigarrow (G \vee H))$

lemma LT12: $\vdash (F \rightsquigarrow H) \longrightarrow (F \rightsquigarrow (G \vee H))$

lemma LT13: $\vdash (G \rightsquigarrow H) \wedge (F \rightsquigarrow G) \longrightarrow (F \rightsquigarrow H)$

lemma LT14: $\vdash ((F \vee G) \rightsquigarrow H) \longrightarrow (F \rightsquigarrow H)$

lemma LT15: $\vdash ((F \vee G) \rightsquigarrow H) \longrightarrow (G \rightsquigarrow H)$

lemma LT16: $\vdash (F \rightsquigarrow H) \wedge (G \rightsquigarrow H) \longrightarrow ((F \vee G) \rightsquigarrow H)$

lemma LT17: $\vdash ((F \vee G) \rightsquigarrow H) = ((F \rightsquigarrow H) \wedge (G \rightsquigarrow H))$

lemma LT18: $\vdash (A \rightsquigarrow (B \vee C)) \wedge (B \rightsquigarrow D) \wedge (C \rightsquigarrow D) \longrightarrow (A \rightsquigarrow D)$

lemma LT19: $\vdash (A \rightsquigarrow (D \vee B)) \wedge (B \rightsquigarrow D) \longrightarrow (A \rightsquigarrow D)$

lemma LT20: $\vdash (A \rightsquigarrow (B \vee D)) \wedge (B \rightsquigarrow D) \longrightarrow (A \rightsquigarrow D)$

lemma LT21: $\vdash ((\exists\, x.\ F\ x) \rightsquigarrow G) = (\forall\, x.\ (F\ x \rightsquigarrow G))$

lemma LT22: $\vdash (F \rightsquigarrow (G \vee H)) \longrightarrow \neg\Diamond G \longrightarrow (F \rightsquigarrow H)$

lemma LT23: $\vdash\!\!\sim (P \longrightarrow \circ Q) \longrightarrow (P \longrightarrow \Diamond Q)$

lemma LT24: $\vdash \Box I \wedge ((P \wedge I) \rightsquigarrow Q) \longrightarrow P \rightsquigarrow Q$

lemma LT25: $\vdash (F \rightsquigarrow \#False) = (\Box\neg F)$

lemma LT26: **assumes:** $\vdash \Diamond F \longrightarrow \Box\Diamond F$ **shows:** $\vdash F \longrightarrow \Box\Diamond F$

lemma LT28: **assumes:** $\vdash\!\!\sim P \longrightarrow \circ P \vee \circ Q$ **shows:** $\vdash\!\!\sim (P \longrightarrow \circ P) \vee \Diamond Q$

lemma LT29: **assumes:** $\vdash\!\!\sim P \longrightarrow \circ P \vee \circ Q$ **and** $\vdash\!\!\sim P \wedge \mathrm{Unchanged}\ v \longrightarrow \circ P$
shows: $\vdash P \longrightarrow \Box P \vee \Diamond Q$

lemma LT30:
assumes: $\vdash\!\!\sim N \wedge P \longrightarrow \circ P \vee \circ Q$ **shows:** $\vdash\!\!\sim N \longrightarrow (P \longrightarrow \circ P) \vee \Diamond Q$

lemma LT31: **assumes:** $\vdash\!\!\sim N \wedge P \longrightarrow \circ P \vee \circ Q$ **and** $\vdash\!\!\sim P \wedge \mathrm{Unchanged}\ v \longrightarrow \circ P$
shows: $\vdash \Box N \longrightarrow P \longrightarrow \Box P \vee \Diamond Q$

lemma LT32:
assumes: $\vdash\!\!\sim (P \wedge [N]\_f) \longrightarrow \circ P \vee \circ Q$ **and** $\vdash\!\!\sim P \wedge \mathrm{Unchanged}\ f \longrightarrow \circ P$
shows: $\vdash \Box[N]\_f \longrightarrow P \longrightarrow \Box P \vee \Diamond Q$

lemma AA1: $\vdash \Box[\#False]\_v \longrightarrow \neg\Diamond\langle Q\rangle\_v$

lemma AA2: $\vdash \Box[P]\_v \wedge \Diamond\langle Q\rangle\_v \longrightarrow \Diamond\langle P\rangle\_v$

lemma AA3: $\vdash \Box[P \longrightarrow Q]\_v \wedge \Box P \wedge \Diamond\langle A\rangle\_v \longrightarrow \Diamond Q$

lemma AA5: $\vdash \Box[P \longrightarrow Q]\_v \longrightarrow \Diamond\langle P\rangle\_v \longrightarrow \Diamond\langle Q\rangle\_v$

lemma AA6: $\vdash \Box[P \longrightarrow Q]\_v \wedge \Diamond\langle P\rangle\_v \longrightarrow \Diamond\langle Q\rangle\_v$

lemma AA7: **assumes:** $\vdash\!\!\sim P \longrightarrow Q$ **shows:** $\vdash \Diamond\langle P\rangle\_v \longrightarrow \Diamond\langle Q\rangle\_v$

lemma AA8: $\vdash \Box[P]\_v \wedge \Diamond\langle A\rangle\_v \longrightarrow \Diamond\langle \Box[P]\_v \wedge A\rangle\_v$

lemma AA9: $\vdash \Box[P]\_v \wedge \Diamond\langle A\rangle\_v \longrightarrow \Diamond\langle [P]\_v \wedge A\rangle\_v$

lemma AA10: $\vdash \neg(\Box[P]\_v \wedge \Diamond\langle \neg P\rangle\_v)$

lemma AA11: $\vdash \neg\Diamond\langle v\$ = \$v\rangle_v$

lemma AA12: $\vdash (\Box[P]_v) = (\neg\Diamond\langle\neg P\rangle_v)$

lemma AA13: $\vdash \Diamond\langle P\rangle_v \longrightarrow \Diamond\langle v\$ \neq \$v\rangle_v$

lemma AA14: $\vdash (\Diamond\langle P \lor Q\rangle_v) = (\Diamond\langle P\rangle_v \lor \Diamond\langle Q\rangle_v)$

lemma AA15: $\vdash \Diamond\langle P \land Q\rangle_v \longrightarrow \Diamond\langle P\rangle_v$

lemma AA16: $\vdash \Diamond\langle P \land Q\rangle_v \longrightarrow \Diamond\langle Q\rangle_v$

lemma AA17: $\vdash \Diamond\langle [P]_v \land A\rangle_v \longrightarrow \Diamond\langle P \land A\rangle_v$

lemma AA18: $\vdash \Box[P]_v \land \Diamond\langle A\rangle_v \longrightarrow \Diamond\langle P \land A\rangle_v$

lemma AA19: $\vdash \Box P \land \Diamond\langle A\rangle_v \longrightarrow \Diamond\langle P \land A\rangle_v$

lemma AA20: **assumes:** $\vdash\!\!\sim P \longrightarrow \circ P \lor \circ Q$ **and** $\vdash\!\!\sim P \land A \longrightarrow \circ Q$
and $\vdash\!\!\sim P \land \text{Unchanged } w \longrightarrow \circ P$
shows: $\vdash \Box(\Box P \longrightarrow \Diamond\langle A\rangle_v) \longrightarrow (P \rightsquigarrow Q)$

lemma AA21: $\vdash\!\!\sim \Diamond\langle \circ F\rangle_v \longrightarrow \circ\Diamond F$

lemma AA22: **assumes:** $\vdash\!\!\sim N \land P \longrightarrow \circ P \lor \circ Q$ **and** $\vdash\!\!\sim (N \land P) \land A \longrightarrow \circ Q$
and $\vdash\!\!\sim P \land \text{Unchanged } w \longrightarrow \circ P$
shows: $\vdash \Box N \land \Box(\Box P \longrightarrow \Diamond\langle A\rangle_v) \longrightarrow (P \rightsquigarrow Q)$

lemma AA23: **assumes:** $\vdash\!\!\sim N \land P \longrightarrow \circ P \lor \circ Q$ **and** $\vdash\!\!\sim (N \land P) \land A \longrightarrow \circ Q$
and $\vdash\!\!\sim P \land \text{Unchanged } w \longrightarrow \circ P$
shows: $\vdash \Box N \land \Box\Diamond\langle A\rangle_v \longrightarrow (P \rightsquigarrow Q)$

lemma AA24: $\vdash (\Diamond\langle\langle P\rangle_f\rangle_f) = (\Diamond\langle P\rangle_f)$

lemma AA25: **assumes:** $\vdash\!\!\sim \langle P\rangle_v \longrightarrow \langle Q\rangle_w$ **shows:** $\vdash \Diamond\langle P\rangle_v \longrightarrow \Diamond\langle Q\rangle_w$

lemma AA26: $\vdash (\Diamond\Box[\neg P]_v) = (\neg\Box\Diamond\langle P\rangle_v)$

lemma AA27: $\vdash (\neg\Diamond\Box[\neg P]_v) = (\Box\Diamond\langle P\rangle_v)$

lemma AA28: $\vdash (\Diamond\langle A\rangle_v) = (\Diamond\Diamond\langle A\rangle_v)$

lemma AA29: $\vdash \Box[N]_v \land \Box\Diamond\langle A\rangle_v \longrightarrow \Box\Diamond\langle N \land A\rangle_v$

lemma AA30: $\vdash (\Diamond\langle\Diamond\langle P\rangle_f\rangle_f) = (\Diamond\langle P\rangle_f)$

lemma AA31: $\vdash \Diamond\langle \circ F\rangle_v \longrightarrow \Diamond F$

lemma log1: **assumes:** $(\vdash P) = (\vdash Q)$ **shows:** $(\vdash \Box P) = (\vdash \Box Q)$

lemma nex_pax2: **assumes:** $\vdash F \longrightarrow G$ **shows:** $\vdash\!\!\sim \circ F \longrightarrow \circ G$

lemma next_and: $\vdash\!\!\sim (\circ(F \land G)) = (\circ F \land \circ G)$

lemma next_or: $\vdash\!\!\sim (\circ(F \lor G)) = (\circ F \lor \circ G)$

lemma next_imp: $\vdash\!\!\sim (\circ(F \longrightarrow G)) = (\circ F \longrightarrow \circ G)$

lemma next_not: $\vdash\!\!\sim (\circ\neg F) = (\neg\circ F)$

lemma next_eq: $\vdash\!\!\sim (\circ(F = G)) = ((\circ F) = (\circ G))$

lemma next_noteq: $\vdash\!\!\sim (\circ(F \neq G)) = ((\circ F) \neq (\circ G))$

lemma next_const: $\vdash\!\!\sim (\circ \#c) = \#c$

lemma next_fun1: $\vdash\!\!\sim (\circ f{<}x{>}) = (f{<}\circ x{>})$

lemma `next_fun2:` $\vdash (\circ f\langle x,y\rangle) = (f\langle \circ x, \circ y\rangle)$
lemma `next_fun3:` $\vdash (\circ f\langle x,y,z\rangle) = (f\langle \circ x, \circ y, \circ z\rangle)$
lemma `next_fun4:` $\vdash (\circ f\langle x,y,z,zz\rangle) = (f\langle \circ x, \circ y, \circ z, \circ zz\rangle)$
lemma `next_forall:` $\vdash (\circ(\forall\, x.\ P\ x)) = (\forall\, x.\ \circ(P\ x))$
lemma `next_exists:` $\vdash (\circ(\exists\, x.\ P\ x)) = (\exists\, x.\ \circ(P\ x))$
lemma `next_exists1:` $\vdash (\circ(\exists!\, x.\ P\ x)) = (\exists!\, x.\ \circ(P\ x))$
lemma `next_le:` $\vdash (\circ(x < y)) = ((\circ x) < (\circ y))$
lemma `next_le_eq:` $\vdash (\circ(x \leq y)) = ((\circ x) \leq (\circ y))$
lemma `next_mem:` $\vdash (\circ(x \in X)) = ((\circ x) \in (\circ X))$
lemma `next_notmem:` $\vdash (\circ(x \notin X)) = ((\circ x) \notin (\circ X))$
lemma `next_if:` $\vdash (\circ(\text{if } T \text{ then } A \text{ else } B)) = (\text{if } \circ T \text{ then } \circ A \text{ else } \circ B)$
lemma `next_plus:` $\vdash (\circ(x + y)) = ((\circ x) + (\circ y))$
lemma `next_minus:` $\vdash (\circ(x - y)) = ((\circ x) - (\circ y))$
lemma `next_times:` $\vdash (\circ(x * y)) = ((\circ x) * (\circ y))$
lemma `next_div:` $\vdash (\circ(x \text{ div } y)) = ((\circ x) \text{ div } (\circ y))$
lemma `next_mod:` $\vdash (\circ(x \text{ mod } y)) = ((\circ x) \text{ mod } (\circ y))$
lemma `next_finset:` $\vdash (\circ X) = \circ X$
lemma `next_pair:` $\vdash (\circ(F,G)) = (\circ F, \circ G)$
lemma `next_cons:` $\vdash (\circ(x\#xs)) = ((\circ x)\#(\circ xs))$
lemma `next_app:` $\vdash (\circ(xs@ys)) = ((\circ xs)@(\circ ys))$
lemma `next_list:` $\vdash (\circ[L]) = [\circ L]$
lemma `next_if:` $\vdash (\circ(\text{if } A \text{ then } B \text{ else } C) = (\text{if } \circ A \text{ then } \circ B \text{ else } \circ C)$
lemma `next_always:` $\vdash \Box F \longrightarrow \circ\Box F$
lemma `next_ev:` **assumes:** stutinv F **shows:** $\vdash \Diamond F \longrightarrow \neg F \longrightarrow \circ\Diamond F$
lemma `next_action:` $\vdash \Box[P]\_v \longrightarrow \circ\Box[P]\_v$

Higher level derived rules

theorem `TLA1:` **assumes:** $\vdash P \wedge \text{Unchanged } f \longrightarrow \circ P$
shows: $\vdash (\Box P) = (P \wedge \Box[P \longrightarrow \circ P]\_f)$
theorem `TLA2:`
assumes: $\vdash P \longrightarrow Q$ **and** $\vdash ([A]\_f) \longrightarrow ([B]\_g)$
shows: $\vdash \Box P \wedge \Box[A]\_f \longrightarrow \Box Q \wedge \Box[B]\_g$
theorem `mTLA2:` **assumes:** $\vdash ([A]\_f) \longrightarrow [B]\_g$ **shows:** $\vdash \Box[A]\_f \longrightarrow \Box[B]\_g$
theorem `mTLA2_split:` **assumes:** $\vdash A \longrightarrow [B]\_g$ **and** $\vdash \text{Unchanged } f \longrightarrow [B]\_g$
shows: $\vdash ([A]\_f) \longrightarrow [B]\_g$
theorem `INV1:` **assumes:** $\vdash I \wedge [N]\_f \longrightarrow \circ I$ **shows:** $\vdash I \wedge \Box[N]\_f \longrightarrow \Box I$
theorem `INV2:` $\vdash \Box I \longrightarrow ((\Box[N]\_f) = (\Box[N \wedge I \wedge \circ I]\_f))$

theorem `R1`: **assumes:** $\vdash\!\!\sim$ Unchanged w $\longrightarrow$ Unchanged v
shows: $\vdash \Box[F]\_w \longrightarrow \Box[F]\_v$

theorem `R2`:
assumes: $\vdash\!\!\sim$ Unchanged w $\longrightarrow$ Unchanged v **and** $\vdash \Box[P]\_w$ **and** $\vdash \Box[P \longrightarrow Q]\_w$
shows: $\vdash \Box[Q]\_v$

theorem `act1`:
assumes: $\vdash\!\!\sim$ Unchanged w $\longrightarrow$ Unchanged v **and** $\vdash \Box[(P \longrightarrow [Q]\_v)]\_w$
shows: $\vdash \Box[P]\_w \longrightarrow \Box[Q]\_v$

theorem `invmono`: **assumes:** $\vdash I \longrightarrow P$ **and** $\vdash\!\!\sim P \wedge [N]\_f \longrightarrow \circ P$
shows: $\vdash I \wedge \Box[N]\_f \longrightarrow \Box P$

theorem `preimpsplit`: **assumes:** $\vdash\!\!\sim I \wedge N \longrightarrow Q$ **and** $\vdash\!\!\sim I \wedge$ Unchanged v $\longrightarrow Q$
shows: $\vdash\!\!\sim I \wedge ([N]\_v) \longrightarrow Q$

theorem `preinvsplit`: **assumes:** $\vdash\!\!\sim I \wedge N \longrightarrow \circ I$ **and** $\vdash\!\!\sim I \wedge$ Unchanged v $\longrightarrow \circ I$
shows: $\vdash\!\!\sim I \wedge [N]\_v \longrightarrow \circ I$

theorem `refinement1`: **assumes:** $\vdash P \longrightarrow Q$ **and** $\vdash\!\!\sim ([A]\_f) \longrightarrow [B]\_g$
shows: $\vdash P \wedge \Box[A]\_f \longrightarrow Q \wedge \Box[B]\_g$

theorem `refstep`:
assumes: $\vdash\!\!\sim$ Unchanged v $\longrightarrow$ Unchanged w **and** $\vdash\!\!\sim P \longrightarrow Q \vee$ Unchanged w
shows: $\vdash\!\!\sim ([P]\_v) \longrightarrow [Q]\_w$

theorem `spec_inv2_mono`:
assumes: $\vdash I \wedge \Box[N]\_v \longrightarrow \Box J$ **and** $\vdash I \wedge \Box[N \wedge J \wedge \circ J]\_v \longrightarrow P$
shows: $\vdash I \wedge \Box[N]\_v \longrightarrow P$

theorem `inv_join`: **assumes:** $\vdash P \longrightarrow \Box Q$ **and** $\vdash P \longrightarrow \Box R$
shows: $\vdash P \longrightarrow \Box(Q \wedge R)$

lemma `inv_case`:
assumes: $\vdash P \longrightarrow \Box(A \longrightarrow B)$ **and** $\vdash P \longrightarrow \Box(\neg A \longrightarrow B)$
shows: $\vdash P \longrightarrow \Box B$

A.1.6 Liveness

Properties about **Enabled**

lemma `EnabledI`: $\vdash F \longrightarrow$ Enabled F

lemma `Enabled_mono`:
assumes: w $\vdash$ Enabled F **and** $\vdash F \longrightarrow G$ **shows:** w $\vDash$ Enabled G

lemma `Enabled_disj1`: $\vdash$ (Enabled F $\longrightarrow$ Enabled (F $\vee$ G))

lemma `Enabled_disj2`: $\vdash$ (Enabled F $\longrightarrow$ Enabled (G $\vee$ F))

lemma `Enabled_conj1`: $\vdash$ (Enabled (F $\wedge$ G) $\longrightarrow$ Enabled F)

lemma `Enabled_conj2:` $\vdash$ (Enabled (G $\wedge$ F) $\longrightarrow$ Enabled F)
lemma `Enabled_conjE:`
assumes: w $\vDash$ Enabled (F $\wedge$ G) **and** $[\![$w $\vDash$ Enabled F ; w $\vDash$ Enabled G $]\!]\longrightarrow$ Q
shows: Q
lemma `Enabled_disjD:` $\vdash$ Enabled (F $\vee$ G) $\longrightarrow$ (Enabled F) $\vee$ (Enabled G)
lemma `Enabled_disj:` $\vdash$ Enabled (F $\vee$ G) = ((Enabled F) $\vee$ (Enabled G))
lemma `Enabled_ex:` $\vdash$ Enabled ($\exists$ x. F x) = ($\exists$ x. Enabled (F x))

Properties about **WF** and **SF**

lemma `WF_alt:` $\vdash$ (WF(A)_v) = (($\Box\Diamond\neg$Enabled ($\langle$A$\rangle$_v)) $\vee$ $\Box\Diamond\langle$A$\rangle$_v)
lemma `SF_alt:` $\vdash$ (SF(A)_v) = (($\Diamond\Box\neg$Enabled ($\langle$A$\rangle$_v)) $\vee$ $\Box\Diamond\langle$A$\rangle$_v)
lemma `AlwaysWFI:` $\vdash$ WF(A)_v $\longrightarrow$ $\Box$WF(A)_v
lemma `WF_always:` $\vdash$ ($\Box$WF(A)_v) = (WF(A)_v)
lemma `AlwaysSFI:` $\vdash$ SF (A)_v $\longrightarrow$ $\Box$SF (A)_v
lemma `SF_always:` $\vdash$ ($\Box$SF(A)_v) = (SF(A)_v)
lemma `Enabled_WFSF:` $\vdash$ $\Box$Enabled ($\langle$ F $\rangle$_v) $\longrightarrow$ ((WF(F)_v) = (SF(F)_v))
lemma `WF1_old:`
assumes: $\mathrel{|\!\sim}$ N $\wedge$ P $\longrightarrow$ $\circ$P $\vee$ $\circ$Q **and** $\mathrel{|\!\sim}$ (N $\wedge$ P) $\wedge$ A $\longrightarrow$ $\circ$Q
and $\vdash$ P $\wedge$ N $\longrightarrow$ Enabled ($\langle$A$\rangle$_v) **and** $\mathrel{|\!\sim}$ P $\wedge$ Unchanged w $\longrightarrow$ $\circ$P
shows: $\vdash$ $\Box$N $\wedge$ (WF(A)_v) $\longrightarrow$ (P $\leadsto$ Q)
theorem `WF1:`
assumes: $\mathrel{|\!\sim}$ P $\wedge$ [N]_f $\longrightarrow$ $\circ$P $\vee$ $\circ$Q **and** $\mathrel{|\!\sim}$ P $\wedge$ $\langle$N $\wedge$ A$\rangle$_f $\longrightarrow$ $\circ$Q
and $\vdash$ P $\longrightarrow$ (Enabled ($\langle$A$\rangle$_f)) **and** $\mathrel{|\!\sim}$ P $\wedge$ Unchanged f $\longrightarrow$ $\circ$P
shows: $\vdash$ $\Box$[N]_f $\wedge$ (WF(A)_f) $\longrightarrow$ (P $\leadsto$ Q)
theorem `SF1:`
assumes: $\mathrel{|\!\sim}$ (P $\wedge$ [N]_f) $\longrightarrow$ ($\circ$P $\vee$ $\circ$Q) **and** $\mathrel{|\!\sim}$ (P $\wedge$ $\langle$(N $\wedge$ A)$\rangle$_f) $\longrightarrow$ $\circ$Q
and $\vdash$ ($\Box$P $\wedge$ $\Box$[N]_f $\wedge$ $\Box$F) $\longrightarrow$ $\Diamond$Enabled ($\langle$A$\rangle$_f)
and $\mathrel{|\!\sim}$ P $\wedge$ Unchanged f $\longrightarrow$ $\circ$P
shows: $\vdash$ ($\Box$[N]_f $\wedge$ (SF(A)_f) $\wedge$ $\Box$F) $\longrightarrow$ (P $\leadsto$ Q)
theorem `WF2:`
assumes: $\mathrel{|\!\sim}$ $\langle$(N $\wedge$ B)$\rangle$_f $\longrightarrow$ $\langle$M$\rangle$_g **and** $\mathrel{|\!\sim}$ P $\wedge$ $\circ$P $\wedge$ $\langle$(N $\wedge$ A)$\rangle$_f $\longrightarrow$ B
and $\vdash$ P $\wedge$ Enabled ($\langle$M$\rangle$_g) $\longrightarrow$ Enabled ($\langle$A$\rangle$_f)
and $\vdash$ $\Box$[(N $\wedge$ $\neg$B)]_f $\wedge$ (WF(A)_f) $\wedge$ $\Box$F
$\wedge$ ($\Diamond\Box$Enabled ($\langle$M$\rangle$_g)) $\longrightarrow$ $\Diamond\Box$P
shows: $\vdash$ $\Box$[N]_f $\wedge$ (WF(A)_f) $\wedge$ $\Box$F $\longrightarrow$ WF(M)_g
theorem `SF2:`
assumes: $\mathrel{|\!\sim}$ $\langle$(N $\wedge$ B)$\rangle$_f $\longrightarrow$ $\langle$M$\rangle$_g **and** $\mathrel{|\!\sim}$ P $\wedge$ $\circ$P $\wedge$ $\langle$(N $\wedge$ A)$\rangle$_f $\longrightarrow$ B

and $\vdash P \wedge \text{Enabled}\ (\langle M \rangle\_g) \longrightarrow \text{Enabled}\ (\langle A \rangle\_f)$
and $\vdash \Box[(N \wedge \neg B)]\_f \wedge (SF(A)\_f) \wedge \Box F$
$\wedge\ (\Box\Diamond\text{Enabled}\ (\langle M \rangle\_g)) \longrightarrow \Diamond\Box P$
shows: $\vdash \Box[N]\_f \wedge (SF(A)\_f) \wedge \Box F \longrightarrow SF(M)\_g$

Various liveness properties

theorem `wf_leadsto`:
assumes: wf r **and** $\forall$ x. $w \vDash F\ x \rightsquigarrow (G \vee (\exists\ y.\ \#((y,x) \in r) \wedge F\ y))$
shows: $w \vDash F\ x \rightsquigarrow G$
lemma `stut_Enabled`: STUTINV (Enabled $(\langle F \rangle\_v)$)
lemma `stut_WF`: **assumes:** NSTUTINV F **shows:** STUTINV WF(F)_v
lemma `stut_SF`: **assumes:** NSTUTINV F **shows:** STUTINV SF(F)_v
lemma `live_invmono`: **assumes:** $\vdash I \longrightarrow P$ **and** $\vdash\!\!\sim P \wedge [N]\_f \longrightarrow \circ P$
shows: $\vdash I \wedge \Box[N]\_f \wedge \text{Live} \longrightarrow \Box P$
theorem `live_refinementmono`:
assumes: $\vdash P \longrightarrow Q$ **and** $\vdash\!\!\sim ([A]\_f) \longrightarrow [B]\_g$ **and** $\vdash P \wedge \Box[A]\_f \wedge L1 \longrightarrow L2$
shows: $\vdash P \wedge \Box[A]\_f \wedge L1 \longrightarrow Q \wedge \Box[B]\_g \wedge L2$
theorem `strenghen_live_invmono`:
assumes: $\vdash I \wedge J \longrightarrow P$ **and** $\vdash\!\!\sim P \wedge [N \wedge J \wedge \circ J]\_f \longrightarrow \circ P$
and $\vdash I \wedge \Box[N]\_f \wedge \text{Live} \longrightarrow \Box J$
shows: $\vdash I \wedge \Box[N]\_f \wedge \text{Live} \longrightarrow \Box P$

A.1.7 State

lemma `basevars`: $\bigwedge$ vs. basevars vs $\longrightarrow \exists$ u. vs u = c
lemma `basevars_range`: $\bigwedge$ vs. range vs = UNIV $\longrightarrow \exists$ u. vs u = c
lemma `bpair1`: $\bigwedge$(x :: state $\Rightarrow$ 'b) y :: state $\Rightarrow$ 'c.
range (LIFT(x, y)) = UNIV $\longrightarrow$ UNIV $\subseteq$ range x
lemma `base_pair1`: basevars (x,y) $\longrightarrow$ basevars x
lemma `bpair2`: $\bigwedge$(x :: state $\Rightarrow$ 'a) y :: state $\Rightarrow$ 'b.
range (LIFT(x, y)) = UNIV $\longrightarrow$ UNIV $\subseteq$ range y
lemma `base_pair2`: basevars (x,y) $\longrightarrow$ basevars y
lemma `base_pair`: basevars (x,y) $\longrightarrow$ basevars x $\wedge$ basevars y
lemma `base_pair`: $\wedge$ x y. basevars (x,y) $\longrightarrow$ basevars x $\wedge$ basevars y
lemma `unit_base`: basevars (v::state $\Rightarrow$ unit)
lemma `baseE`: **assumes:** basevars v **and** $\wedge$ x. v x = c $\longrightarrow$ Q **shows:** Q

theorem `base_enabled`:

assumes: basevars vs **and** ∃ c. ∀ u. vs (first u) = c ⟶ (((first s) ## u) ⊢ F)

shows: s ⊨ Enabled F

A.2 The Hume mechanisation

Tuple projection

lemma `tup_proj2`: (x = (a,b)) = (fst x = a ∧ snd x = b)

lemma `tup_proj3`: (x = (a,b,c)) = (fst3 x = a ∧ snd3 x = b ∧ thd3 x = c)

lemma `tup_proj4`: (x = (a,b,c,d)) = (fst4 x = a ∧ snd4 x = b ∧ thd4 x = c ∧ for4 x = d)

lemma `tup_proj5`: (x = (a,b,c,d,e)) = (fst5 x = a ∧ snd5 x = b ∧ thd5 x = c ∧ for5 x = d ∧ fif5 x = e)

lemma `tup_proj6`: (x = (a,b,c,d,e,f)) = (fst6 x = a ∧ snd6 x = b ∧ thd6 x = c ∧ for6 x = d ∧ fif6 x = e ∧ six6 x = f)

lemma `tup_proj7`: (x = (a,b,c,d,e,f,g)) = (fst7 x = a ∧ snd7 x = b ∧ thd7 x = c ∧ for7 x = d ∧ fif7 x = e ∧ six7 x = f ∧ sev7 x = g)

lemma `tup_proj8`: (x = (a,b,c,d,e,f,g,h)) = (fst8 x = a ∧ snd8 x = b ∧ thd8 x = c ∧ for8 x = d ∧ fif8 x = e ∧ six8 x = f ∧ sev8 x = g ∧ eig8 x= h)

lemma `tup_proj9`: (x = (a,b,c,d,e,f,g,h,i)) = (fst9 x = a ∧ snd9 x = b ∧ thd9 x = c ∧ for9 x = d ∧ fif9 x = e ∧ six9 x = f ∧ sev9 x = g ∧ eig9 x= h ∧ nin9 x= i)

lemma `tup_proj10`: (x = (a,b,c,d,e,f,g,h,i,j)) = (fst10 x = a ∧ snd10 x = b ∧ thd10 x = c ∧ for10 x = d ∧ fif10 x = e ∧ six10 x = f ∧ sev10 x = g ∧ eig10 x= h ∧ nin10 x= i ∧ ten10 x = j)

lemma `tup_proj11`: (x = (a,b,c,d,e,f,g,h,i,j,k)) = (fst11 x = a ∧ snd11 x = b ∧ thd11 x = c ∧ for11 x = d ∧ fif11 x = e ∧ six11 x = f ∧ sev11 x = g ∧ eig11 x= h ∧ nin11 x= i ∧ ten11 x = j ∧ ele11 x = k)

lemma `tup_lup1`: fst (x,y) = x

lemma `tup_lup2`: snd (x,y) = y

lemma `tup_lup3`: fst3 (x,y,z) = x

lemma `tup_lup4`: snd3 (x,y,z) = y

lemma `tup_lup5`: thd3 (x,y,z) = z

lemma `tup_lup6`: fst4 (a,b,c,d) = a

lemma `tup_lup7`: snd4 (a,b,c,d) = b

lemma `tup_lup8`: thd4 (a,b,c,d) = c

lemma tup_lup9: for4 (a,b,c,d) = d
lemma tup_lup10: fst5 (a,b,c,d,e) = a
lemma tup_lup11: snd5 (a,b,c,d,e) = b
lemma tup_lup12: thd5 (a,b,c,d,e) = c
lemma tup_lup13: for5 (a,b,c,d,e) = d
lemma tup_lup14: fif5 (a,b,c,d,e) = e
lemma tup_lup15: fst6 (a,b,c,d,e,f) = a
lemma tup_lup16: snd6 (a,b,c,d,e,f) = b
lemma tup_lup17: thd6 (a,b,c,d,e,f) = c
lemma tup_lup18: for6 (a,b,c,d,e,f) = d
lemma tup_lup19: fif6 (a,b,c,d,e,f) = e
lemma tup_lup20: six6 (a,b,c,d,e,f) = f
lemma tup_lup21: fst7 (a,b,c,d,e,f,g) = a
lemma tup_lup22: snd7 (a,b,c,d,e,f,g) = b
lemma tup_lup23: thd7 (a,b,c,d,e,f,g) = c
lemma tup_lup24: for7 (a,b,c,d,e,f,g) = d
lemma tup_lup25: fif7 (a,b,c,d,e,f,g) = e
lemma tup_lup26: six7 (a,b,c,d,e,f,g) = f
lemma tup_lup27: sev7 (a,b,c,d,e,f,g) = g
lemma tup_lup28: fst8 (a,b,c,d,e,f,g,h) = a
lemma tup_lup29: snd8 (a,b,c,d,e,f,g,h) = b
lemma tup_lup30: thd8 (a,b,c,d,e,f,g,h) = c
lemma tup_lup31: for8 (a,b,c,d,e,f,g,h) = d
lemma tup_lup32: fif8 (a,b,c,d,e,f,g,h) = e
lemma tup_lup33: six8 (a,b,c,d,e,f,g,h) = f
lemma tup_lup34: sev8 (a,b,c,d,e,f,g,h) = g
lemma tup_lup35: eig8 (a,b,c,d,e,f,g,h) = h
lemma tup_lup36: fst9 (a,b,c,d,e,f,g,h,i) = a
lemma tup_lup37: snd9 (a,b,c,d,e,f,g,h,i) = b
lemma tup_lup38: thd9 (a,b,c,d,e,f,g,h,i) = c
lemma tup_lup39: for9 (a,b,c,d,e,f,g,h,i) = d
lemma tup_lup40: fif9 (a,b,c,d,e,f,g,h,i) = e
lemma tup_lup41: six9 (a,b,c,d,e,f,g,h,i) = f
lemma tup_lup42: sev9 (a,b,c,d,e,f,g,h,i) = g
lemma tup_lup43: eig9 (a,b,c,d,e,f,g,h,i) = h
lemma tup_lup44: nin9 (a,b,c,d,e,f,g,h,i) = i
lemma tup_lup45: fst10 (a,b,c,d,e,f,g,h,i,j) = a

lemma `tup_lup46:` snd10 (a,b,c,d,e,f,g,h,i,j) = b
lemma `tup_lup47:` thd10 (a,b,c,d,e,f,g,h,i,j) = c
lemma `tup_lup48:` for10 (a,b,c,d,e,f,g,h,i,j) = d
lemma `tup_lup49:` fif10 (a,b,c,d,e,f,g,h,i,j) = e
lemma `tup_lup50:` six10 (a,b,c,d,e,f,g,h,i,j) = f
lemma `tup_lup51:` sev10 (a,b,c,d,e,f,g,h,i,j) = g
lemma `tup_lup52:` eig10 (a,b,c,d,e,f,g,h,i,j) = h
lemma `tup_lup53:` nin10 (a,b,c,d,e,f,g,h,i,j) = i
lemma `tup_lup54:` ten10 (a,b,c,d,e,f,g,h,i,j) = j
lemma `tup_lup55:` fst11 (a,b,c,d,e,f,g,h,i,j,k) = a
lemma `tup_lup56:` snd11 (a,b,c,d,e,f,g,h,i,j,k) = b
lemma `tup_lup57:` thd11 (a,b,c,d,e,f,g,h,i,j,k) = c
lemma `tup_lup58:` for11 (a,b,c,d,e,f,g,h,i,j,k) = d
lemma `tup_lup59:` fif11 (a,b,c,d,e,f,g,h,i,j,k) = e
lemma `tup_lup60:` six11 (a,b,c,d,e,f,g,h,i,j,k) = f
lemma `tup_lup61:` sev11 (a,b,c,d,e,f,g,h,i,j,k) = g
lemma `tup_lup62:` eig11 (a,b,c,d,e,f,g,h,i,j,k) = h
lemma `tup_lup63:` nin11 (a,b,c,d,e,f,g,h,i,j,k) = i
lemma `tup_lup64:` ten11 (a,b,c,d,e,f,g,h,i,j,k) = j
lemma `tup_lup65:` ele11 (a,b,c,d,e,f,g,h,i,j,k) = k

Hume values

lemma `not_isVal:` ¬(isVal k) ⟹ k = None
lemma `isEmpty_bot:` isEmpty k ⟹ k = None
lemma `isEmpty_bot2:` isEmpty k = (k = None)
lemma `stut_tobeforeval:` STUTINV @f
lemma `nstut_toafterval:` NSTUTINV f@
lemma `t2:` ⊢∼ (∘@x) = (x@)
lemma `not_isVal:` (¬(isVal k)) = (k = None)
lemma `inv_cas:` (A ∧ B) ⟹ I = ((A ⟹ I) ∧ (B ⟹ I))
lemma `isVal_not_None:` isVal v = (v ≠ None)
lemma `isVal_some:` isVal v = (∃ y. v = Some y)
lemma `isVal_full:` isVal v = (v ≠ None ∧ (∃ y. v = Some y))
lemma `isVal_some:` isVal v ⟹ (toVal v = y) = (v = Some y)
lemma `some_isVal:` x = Some a ⟹ isVal x
lemma `valproj1:` ⟦(w ⊨ x$) = (w ⊨ y$)⟧ ⟹ (w ⊨ x@) = (w ⊨ y@)
lemma `valproj2:` ⟦(w ⊨ $x) = (w ⊨ $y)⟧ ⟹ (w ⊨ @x) = (w ⊨ @y)

lemma valproj3: $[\![(w \vDash x\$) = (w \vDash \$y)]\!] \Longrightarrow (w \vDash x@) = (w \vDash @y)$

lemma valproj4: $[\![(w \vDash \$x) = (w \vDash y\$)]\!] \Longrightarrow (w \vDash @x) = (w \vDash y@)$

lemma valproj1fn1: $[\![f(w \vDash x\$) = (w \vDash y\$)]\!] \Longrightarrow (w \vDash f\langle x\rangle@) = (w \vDash y@)$

lemma valproj1fn2: $[\![f(w \vDash \$x) = (w \vDash \$y)]\!] \Longrightarrow (w \vDash @f\langle x\rangle) = (w \vDash @y)$

lemma valproj1fn3: $[\![f(w \vDash x\$) = (w \vDash \$y)]\!] \Longrightarrow (w \vDash f\langle x\rangle@) = (w \vDash @y)$

lemma valproj1fn4: $[\![f(w \vDash \$x) = (w \vDash y\$)]\!] \Longrightarrow (w \vDash @f\langle x\rangle) = (w \vDash y@)$

lemma valproj2fn1: $[\![(w \vDash x\$) = f(w \vDash y\$)]\!] \Longrightarrow (w \vDash x@) = (w \vDash f\langle y\rangle@)$

lemma valproj2fn2: $[\![(w \vDash \$x) = f(w \vDash \$y)]\!] \Longrightarrow (w \vDash @x) = (w \vDash @f\langle y\rangle)$

lemma valproj2fn3: $[\![(w \vDash x\$) = f(w \vDash \$y)]\!] \Longrightarrow (w \vDash x@) = (w \vDash @f\langle y\rangle)$

lemma valproj2fn4: $[\![(w \vDash \$x) = f(w \vDash y\$)]\!] \Longrightarrow (w \vDash @x) = (w \vDash f\langle y\rangle@)$

lemma valprojfn1: $[\![g(w \vDash x\$) = f(w \vDash y\$)]\!] \Longrightarrow (w \vDash g\langle x\rangle@) = (w \vDash f\langle y\rangle@)$

lemma valprojfn2: $[\![g(w \vDash \$x) = f(w \vDash \$y)]\!] \Longrightarrow (w \vDash @g\langle x\rangle) = (w \vDash @f\langle y\rangle)$

lemma valprojfn3: $[\![g(w \vDash x\$) = f(w \vDash \$y)]\!] \Longrightarrow (w \vDash g\langle x\rangle@) = (w \vDash @f\langle y\rangle)$

lemma valprojfn4: $[\![g(w \vDash \$x) = f(w \vDash y\$)]\!] \Longrightarrow (w \vDash @g\langle x\rangle) = (w \vDash f\langle y\rangle@)$

lemma isvalproj1: $[\![isVal(w \vDash x\$);isVal(w \vDash y\$)]\!]$
$\Longrightarrow ((w \vDash x\$) = (w \vDash y) = ((w \vDash x@) = (w \vDash y@))$

lemma isvalproj2: $[\![isVal(w \vDash \$x);isVal(w \vDash \$y)]\!]$
$\Longrightarrow ((w \vDash \$x) = (w \vDash \$y)) = ((w \vDash @x) = (w \vDash @y))$

lemma isvalproj3: $[\![isVal(w \vDash x\$);isVal(w \vDash \$y)]\!]$
$\Longrightarrow ((w \vDash x\$) = (w \vDash \$y)) = ((w \vDash x@) = (w \vDash @y))$

lemma isvalproj4: $[\![isVal(w \vDash \$x);isVal(w \vDash y\$)]\!]$
$\Longrightarrow ((w \vDash \$x) = (w \vDash y\$)) = ((w \vDash @x) = (w \vDash y@))$

lemma isvalproj1fn1: $[\![isVal(f(w \vDash x\$));isVal(w \vDash y\$)]\!]$
$\Longrightarrow (f(w \vDash x\$) = (w \vDash y\$)) = ((w \vDash f\langle x\rangle@) = (w \vDash y@))$

lemma isvalproj1fn2: $[\![isVal(f(w \vDash \$x));isVal(w \vDash \$y)]\!]$
$\Longrightarrow (f(w \vDash \$x) = (w \vDash \$y)) = ((w \vDash @f\langle x\rangle) = (w \vDash @y))$

lemma isvalproj1fn3: $[\![isVal(f(w \vDash x\$));isVal(w \vDash \$y)]\!]$
$\Longrightarrow (f(w \vDash x\$) = (w \vDash \$y)) = ((w \vDash f\langle x\rangle@) = (w \vDash @y))$

lemma isvalproj1fn4: $[\![isVal(f(w \vDash \$x));isVal(w \vDash y\$)]\!]$
$\Longrightarrow (f(w \vDash \$x) = (w \vDash y\$)) = ((w \vDash @f\langle x\rangle) = (w \vDash y@))$

lemma isvalproj2fn1: $[\![isVal(w \vDash x\$);isVal(f(w \vDash y\$))]\!]$
$\Longrightarrow ((w \vDash x\$) = f(w \vDash y\$)) = ((w \vDash x@) = (w \vDash f\langle y\rangle@))$

lemma isvalproj2fn2: $[\![isVal(w \vDash \$x);isVal(f(w \vDash \$y))]\!]$
$\Longrightarrow ((w \vDash \$x) = f(w \vDash \$y)) = ((w \vDash @x) = (w \vDash @f\langle y\rangle))$

lemma isvalproj2fn3: $[\![isVal(w \vDash x\$);isVal(f(w \vDash \$y))]\!]$
$\Longrightarrow ((w \vDash x\$) = f(w \vDash \$y)) = ((w \vDash x@) = (w \vDash @f\langle y\rangle))$

lemma `isvalproj2fn4:` ⟦isVal(w ⊨ $x);isVal(f(w ⊨ y$))⟧
⟹ ((w ⊨ $x) = f(w ⊨ y$))=((w ⊨ @x) = (w ⊨ f<y>@))
lemma `isvalprojfn1:` ⟦isVal(f(w ⊨ x$));isVal(g(w ⊨ y$))⟧
⟹ (f(w ⊨ x$) = g(w ⊨ y$))=((w ⊨ f<x>@) = (w ⊨ g<y>@))
lemma `isvalprojfn2:` ⟦isVal(f(w ⊨ $x));isVal(g(w ⊨ $y))⟧
⟹ (f(w ⊨ $x) = g(w ⊨ $y))=((w ⊨ @f<x>) = (w ⊨ @g<y>))
lemma `isvalprojfn3:` ⟦isVal(f(w ⊨ x$));isVal(g(w ⊨ $y))⟧
⟹ (f(w ⊨ x$) = g(w ⊨ $y))=((w ⊨ f<x>@) = (w ⊨ @g<y>))
lemma `isvalprojfn4:` ⟦isVal(f(w ⊨ $x));isVal(g(w ⊨ y$))⟧
⟹ (f(w ⊨ $x) = g(w ⊨ y$))=((w ⊨ @f<x>) = (w ⊨ g<y>@))
lemma `valin1:` ⟦isVal v ⟧ ⟹ (v = intoVal y) = (toVal v = y)
lemma `valin2:` ⟦isVal (w ⊨ x$) ⟧ ⟹ ((w ⊨ x$) = intoVal v) = ((w ⊨ x@) = v)
lemma `valin3:` ⟦isVal (w ⊨ $x) ⟧ ⟹ ((w ⊨ $x) = intoVal v) = ((w ⊨ @x) = v)
lemma `valin4:` ⟦isVal (w ⊨ x$) ⟧ ⟹ ((w ⊨ x$) = intoVal v) = ((w ⊨ x@) = v)
lemma `valin5:` ⟦isVal (g(w ⊨ x$))⟧
⟹ ((g(w ⊨ x$)) = (intoVal v)) = ((w ⊨ g<x>@) = v)
lemma `valin6:` ⟦isVal (g(w ⊨ $x))⟧
⟹ ((g(w ⊨ $x)) = (intoVal v)) = ((w ⊨ @g<x>) = v)
lemma `valin1':` ⟦isVal v ⟧ ⟹ (v = Some y) = (toVal v = y)
lemma `valin2':` ⟦isVal (w ⊨ x$) ⟧ ⟹ ((w ⊨ x$) = Some v) = ((w ⊨ x@) = v)
lemma `valin3':` ⟦isVal (w ⊨ $x) ⟧ ⟹ ((w ⊨ $x) = Some v) = ((w ⊨ @x) = v)
lemma `valin4':` ⟦isVal (w ⊨ x$) ⟧ ⟹ ((w ⊨ x$) = Some v) = ((w ⊨ x@) = v)
lemma `valin5':` ⟦isVal (g(w ⊨ x$))⟧
⟹ ((g(w ⊨ x$)) = (Some v)) = ((w ⊨ g<x>@) = v)
lemma `valin6':` ⟦isVal (g(w ⊨ $x))⟧
⟹ ((g(w ⊨ $x)) = (Some v)) = ((w ⊨ @g<x>) = v)
lemma `valintoto:` isVal v ⟹ (v = intoVal y) = (toVal v = y)
lemma `valintoto_lift:` ⊨ isVal<x$> ⟹ (x$ = intoVal <v>) = (x@ = v)
lemma `valintoto_unl:` isVal (w ⊨ x$)
⟹ ((w ⊨ x$) = (intoVal v)) = ((w ⊨ x@)= v)
lemma `valintoto_unl2:` isVal (w ⊨ $x)
⟹ ((w ⊨ $x) = (intoVal v)) = ((w ⊨ @x)= v)
lemma `valintoto_unl_fun:` isVal (g(w ⊨ x$))
⟹ ((g(w ⊨ x$)) = (intoVal v)) = ((w ⊨ g<x>@) = v)
lemma `valintoto_unl_fun2:` isVal (g(w ⊨ $x))
⟹ ((g(w ⊨ $x)) = (intoVal v)) = ((w ⊨ @g<x>) = v)
lemma `condsym:` P ⟹ A = B ⟹ P ⟹ B = A

lemma exe_bod: ((if P then Q else R) ⟹ I) =
((P ∧ Q) ⟹ I) ∧ ((¬P ∧ R) ⟹ I))
lemma exe_bod2: ((M ∧ (if P then Q else R)) ⟹ I) =
((((M ∧ P) ∧ Q) ⟹ I) ∧ (((M ∧ ¬P) ∧ R) ⟹ I))
lemma exe_bl: **assumes:** w ⊨ P ⟹ I **and** w ⊨ Q ⟹ I
shows: w ⊨ (if R then P else Q) ⟹ I
lemma unch_opt_val: ⊢∼ x$ = $y ⟹ x@ = @y
lemma after_opt_val: ⊢∼ x$ = y$ ⟹ x@ = y@
lemma before_opt_val: ⊢∼ $x = $y ⟹ @x = @y
lemma unch_opt_fun_unl: g(w ⊨ x$) = g(w ⊨ $y)
⟹ (w ⊨ g<x>@) = (w ⊨ @g<y>)
lemma isVal_valintoto: isVal(w ⊨ $w4) ⟹ (w ⊨ $w4) = Some (w ⊨ @w4)
lemma isEmpty_None: isEmpty v ⟹ v = None

Pattern matching

lemma pmatch1:
assumes: x = PConsume **and** ¬(PMatch x wire) **shows:** isEmpty wire
lemma pmatch2: **assumes:** ¬(PMatch PConsume wire) **shows:** isEmpty wire
lemma pmatch3: **assumes:** PMatch PConsume wire **shows:** isVal wire
lemma pmatch_const1: (mConst A B) = (B = Some A)
lemma pmatch_con1: mCon A = (∃ v. Some v = A)
lemma pmatch_con2: (mCon A)=(isVal A)
lemma lift_m1: ⊢∼ (mConst<A,$B>)=(isVal<$B> ∧ @B = A)
lemma lift_m2: ⊢∼ mCon<$A> = (isVal<$A> ∧ (∃ v. @A = v))
lemma lift_m3: ⊢∼ (mConst<A,B$>)=(isVal<B$> ∧ B@ = A)
lemma lift_m4: ⊢∼ mCon<A$> = (isVal<A$> ∧ (∃ v. A@ = v))

Output wires

lemma assertout1: isEmpty res ⟹ assertOut res wire
lemma assertout2: isEmpty wire ⟹ assertOut res wire
lemma assertout3: assertOut res wire = (isEmpty res ∨ isEmpty wire)
lemma nwireE: ⟦x = nwire A B; x = A ⟹ P; x = B ⟹ P⟧ ⟹ P
lemma nwire1: (nwire a b = a) ∨ (nwire a b = b)
lemma nwire2: nwire x y ≠ x ⟹ nwire x y = y
lemma nw1: isVal x ⟹ nwire x y = x
lemma nw2: ¬isVal x ⟹ nwire x y = y

lemma `nwire1_lift:` (nwire (w ⊨ $a) (w ⊨ $b) = (w ⊨ $a))
∨ (nwire (w ⊨ $a)(w ⊨ $b) = (w ⊨ $b))
lemma `nwire1_lift_fun:` (nwire (w ⊨ g<$a>) (w ⊨ $b) = (w ⊨ g<$a>))
∨ (nwire (w ⊨ g<$a>)(w ⊨ $b) = (w ⊨ $b))

A.3 Hume case studies

A.3.1 Even and odd numbers

theorem `mainth:` ⊢ evenodd ⟶ □((isVal<$w1> ⟶ @w1 ∈ #OddN)
∧ (isVal<$odd_res> ⟶ @odd_res ∈ #OddN)
∧ (isVal<$w2> ⟶ @w2 ∈ #EvenN)
∧ (isVal<$even_res> ⟶ @even_res ∈ #EvenN))
theorem `evenwire:` ⊢ evenodd ⟶ □(isVal<$w2> ⟶ @w2 ∈ #EvenN)
theorem `oddwire:` ⊢ evenodd ⟶ □(isVal<$w1> ⟶ @w1 ∈ #OddN)

A.3.2 The vending machine

lemma `vend_mon1:` ⊢ vending
⟶ □((isVal<snd3<$con_res>> ⟶ #0 ≤ @snd3<con_res>)
∧ (isVal<$w4> ⟶ #0 ≤ @w4))
theorem `vend_money:` ⊢ vending ⟶ □(isVal<$w4> ⟶ #0 ≤ @w4)

A.4 The Hierarchical Hume mechanisation

lemma `sch1:` S b pc t psch ∈ {Super,Execute,Terminated}
lemma `sch2:` S Execute pc t psch ∈ {Super,Execute}
lemma `sch3:` S Super pc t psch ∈ {Terminated,Execute}
lemma `sch4:` (S b pc t psch = Super) ∨ (S b pc t psch = Terminated)
lemma `sch5:` S Super ppc t True = Terminated
lemma `S1sch1:` S1 s pc t ∈ {Super,Execute}
lemma `S1sch2:` S1 Execute False c = Execute
lemma `termcond1:` termcond x = (x ∈ {Terminated,Blocked,Matchfail})

A.5 Hierarchical Hume case studies

A.5.1 The SAFER system

Generic properties

lemma SAFER_sch: $\vdash$ program $\longrightarrow$ □($s ∈ #Super,Execute)

lemma SAFER_inp1: $\vdash$ program $\longrightarrow$ □(isVal<$w18> $\longrightarrow$ @w18 = @fst11<safer_inp>)

lemma SAFER_inp2: $\vdash$ program $\longrightarrow$ □(isVal<$w15> $\longrightarrow$ @w15 = @snd11<safer_inp>)

lemma SAFER_inp3: $\vdash$ program $\longrightarrow$ □(isVal<$w19> $\longrightarrow$ @w19 = @thd11<safer_inp>)

lemma SAFER_inp4: $\vdash$ program $\longrightarrow$ □(isVal<$w20> $\longrightarrow$ @w20 = @for11<safer_inp>)

lemma SAFER_inp5: $\vdash$ program $\longrightarrow$ □(isVal<$w21> $\longrightarrow$ @w21 = @fif11<safer_inp>)

lemma SAFER_inp6: $\vdash$ program $\longrightarrow$ □(isVal<$w1> $\longrightarrow$ @w1 = @six11<safer_inp>)

lemma SAFER_inp7: $\vdash$ program $\longrightarrow$ □(isVal<$w2> $\longrightarrow$ @w2 = @sev11<safer_inp>)

lemma SAFER_inp8: $\vdash$ program $\longrightarrow$ □(isVal<$w3> $\longrightarrow$ @w3 = @eig11<safer_inp>)

lemma SAFER_inp9: $\vdash$ program $\longrightarrow$ □(isVal<$w4> $\longrightarrow$ @w4= @nin11<safer_inp>)

lemma SAFER_inp10: $\vdash$ program $\longrightarrow$ □(isVal<$w5> $\longrightarrow$ @w5 = @ten11<safer_inp>)

lemma SAFER_inp11: $\vdash$ program $\longrightarrow$ □(isVal<$w7> $\longrightarrow$ @w7 = @ele11<safer_inp>)

lemma SAFER_aahprf1: $\vdash$ program $\longrightarrow$
□(isVal<thd11<$aahprf_res>> $\longrightarrow$ @thd11<aahprf_res> = @fst11<safer_inp>)

lemma SAFER_aahprf2: $\vdash$ program $\longrightarrow$ □(isVal<$w25> $\longrightarrow$ @w25 = @fst11<safer_inp>)

lemma SAFER_aahprf3: $\vdash$ program $\longrightarrow$
□(isVal<for11<$aahprf_res>> $\longrightarrow$ @for11<aahprf_res> = @snd11<safer_inp>)

lemma SAFER_aahprf4: $\vdash$ program $\longrightarrow$ □(isVal<$w26> $\longrightarrow$ @w26 = @snd11<safer_inp>)

lemma rot_tran: (X $\neq$ ROT) = (X = TRAN)

Requirement 1

lemma Req1_AAH_command1: $\vdash$ program
$\longrightarrow$ □(isVal<fst3<$aahpof_res>> $\longrightarrow$ @fst3<aahpof_res> = #(ZERO,ZERO,ZERO))

lemma Req1_AAH_command2: $\vdash$ program $\longrightarrow$ □(isVal<$w24>
$\longrightarrow$ @w24 = #(ZERO,ZERO,ZERO))

lemma Req1_GC_ZERO1: $\vdash$ program $\longrightarrow$ □(@six11<safer_inp> = #ZERO
$\wedge$ @sev11<safer_inp> = #ZERO $\wedge$ @eig11<safer_inp> = #ZERO
$\wedge$ @nin11<safer_inp> = #ZERO $\longrightarrow$ isVal<fst3<$gc_res>>
$\longrightarrow$ @fst3<gc_res> = #(ZERO,ZERO,ZERO))

lemma Req1_GC_ZERO2: $\vdash$ program $\longrightarrow$ □(@six11<safer_inp> = #ZERO
$\wedge$ @sev11<safer_inp> = #ZERO $\wedge$ @eig11<safer_inp> = #ZERO

∧ @nin11<safer_inp> = #ZERO ⟶ isVal<snd3<$gc_res>>
⟶ @snd3<gc_res> = #(ZERO,ZERO,ZERO))

lemma Req1_GC_ZERO3: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ⟶ isVal<$w22>
⟶ @w22 = #(ZERO,ZERO,ZERO))

lemma Req1_GC_ZERO4: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ⟶ isVal<$w23>
⟶ @w23 = #(ZERO,ZERO,ZERO))

lemma Req1_IC_ZERO1: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<fst3<$ic_res>> ⟶ @fst3<ic_res> = #(ZERO,ZERO,ZERO))

lemma Req1_IC_ZERO2: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<snd3<$ic_res>> ⟶ @snd3<ic_res> = #(ZERO,ZERO,ZERO))

lemma Req1_IC_ZERO3: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<$w27> ⟶ @w27 = #(ZERO,ZERO,ZERO))

lemma Req1_IC_ZERO4: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<$w29> ⟶ @w29 = #(ZERO,ZERO,ZERO))

lemma Req1_PTC_ZERO1: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<fst3<$ptc_res>> ⟶ @fst3<ptc_res> = #ZERO)

lemma Req1_PTC_ZERO2: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<snd3<$ptc_res>> ⟶ @snd3<ptc_res> = #ZERO)

lemma Req1_PTC_ZERO3: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)

⟶ isVal<thd3<$ptc_res>> ⟶ @thd3<ptc_res> = #ZERO)

lemma Req1_PTC_ZERO4: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<$w33> ⟶ @w33 = #ZERO)

lemma Req1_PTC_ZERO5: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<$w30> ⟶ @w30 = #ZERO)

lemma Req1_PTC_ZERO6: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<$w31> ⟶ @w31 = #ZERO)

lemma Req1_ICF_ZERO1: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ (isVal<fst4<$icf_res>> ⟶ @fst4<icf_res> = #ZERO) ∧ (isVal<snd4<$icf_res>>
⟶ @snd4<icf_res> = #ZERO) ∧ (isVal<thd4<$icf_res>>
⟶ @thd4<icf_res> = #ZERO))

lemma Req1_ICF_ZERO2: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ (isVal<$w34> ⟶ @w34 = #ZERO) ∧ (isVal<$w35> ⟶ @w35 = #ZERO))

lemma Req1_ICF_ZERO3: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ (isVal<$w32> ⟶ @w32 = #ZERO))

lemma Req1_LRUD_ZERO1: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ (isVal<fst<$lrud_res>> ⟶ @fst<lrud_res> = #[])
∧ (isVal<snd<$lrud_res>> ⟶ @snd<lrud_res> = #[]))

lemma Req1_LRUD_ZERO2: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ (isVal<$w36> ⟶ @w36 = #[]) ∧ (isVal<$w37> ⟶ @w37 = #[]))

lemma Req1_BF_ZERO1: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO

∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ (isVal<fst<$bf_res>> ⟶ @fst<bf_res> = #[])
∧ (isVal<snd<$bf_res>> ⟶ @snd<bf_res> = #[]))

lemma Req1_BF_ZERO2: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ (isVal<$w38> ⟶ @w38 = #[]) ∧ (isVal<$w39> ⟶ @w39 = #[]))

lemma Req1_TJ_ZERO1: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ (isVal<$tj_res> ⟶ @tj_res = #[]))

lemma Req1_TJ_ZERO2: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ (isVal<$w41> ⟶ @w41 = #[]))

theorem Req1_main: ⊢ program ⟶ □(@six11<safer_inp> = #ZERO
∧ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO
∧ @nin11<safer_inp> = #ZERO ∧ @fst11<safer_inp> = #(False,False,False)
⟶ isVal<six6<$safer_res>> ⟶ @six6<safer_res> = #[])

Requirement 2

lemma Prop2_IC_ZERO1: ⊢ program ⟶ □(isVal<thd3<$ic_res>>
∧ isVal<fst3<$ic_res>>
⟶ (@thd3<ic_res> = #False ⟶ @fst3<ic_res> = #(ZERO,ZERO,ZERO)))

lemma Prop2_IC_out1: ⊢ program
⟶ □(isVal<thd3<$ic_res>> = isVal<fst3<$ic_res>>)

lemma Prop2_IC_ZERO2: ⊢ program ⟶ □(isVal<$w28> ∧ isVal<$w27>
⟶ (@w28 = #False ⟶ @w27 = #(ZERO,ZERO,ZERO)))

lemma moa1: max_one_axis ZERO ZERO a

lemma moa2: max_one_axis a ZERO ZERO

lemma moa3: max_one_axis ZERO a ZERO

lemma moa4: max_one_axis ZERO ZERO ZERO

lemma Prop2_1_PTC1: ⊢ program ⟶ □(isVal<fst3<$ptc_res>>
∧ isVal<snd3<$ptc_res>> ∧ isVal<thd3<$ptc_res>>
⟶ max_one_axis<@fst3<ptc_res>,@snd3<ptc_res>,@thd3<ptc_res>>)

lemma `Prop2_1_PTC2:` ⊢ program ⟶ □(isVal<fst3<$ptc_res>>
= isVal<snd3<$ptc_res>> ∧ isVal<fst3<$ptc_res>> = isVal<thd3<$ptc_res>>)
theorem `Prop2_main1:` ⊢ program ⟶ □(isVal<$w33> ∧ isVal<$w30>
∧ isVal<$w31> ⟶ max_one_axis<@w33,@w30,@w31>)
lemma `Prop2_2_GC_1:` ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
∧ isVal<fst3<$gc_res>> ⟶ @fst3<gc_res> = (@sev11<safer_inp>,#ZERO,#ZERO))
lemma `Prop2_2_GC_2:` ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN
∧ isVal<fst3<$gc_res>> ⟶ @fst3<gc_res>
= (@sev11<safer_inp>,@eig11<safer_inp>,@six11<safer_inp>))
lemma `Prop2_2_GC_3:` ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
∧ isVal<$w22> ⟶ @w22 = (@sev11<safer_inp>,#ZERO,#ZERO))
lemma `Prop2_2_GC_4:` ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN
∧ isVal<$w22>
⟶ @w22 = (@sev11<safer_inp>,@eig11<safer_inp>,@six11<safer_inp>))
lemma `Prop2_2_IC_1:` ⊢ program ⟶ □(isVal<thd3<$ic_res>> ∧ @thd3<ic_res>
= #False ∧ isVal<fst3<$ic_res>> ⟶ @fst3<ic_res> = #(ZERO,ZERO,ZERO))
lemma `Prop2_2_IC_out1:` ⊢ program ⟶
□(isVal<thd3<$ic_res>> = isVal<fst3<$ic_res>>)
lemma `Prop2_IC_2:` ⊢ program ⟶ □(isVal<$w28> ∧ @w28 = #False
∧ isVal<$w27> ⟶ @w27 = #(ZERO,ZERO,ZERO))
lemma `Prop2_2_IC_3:` ⊢ program ⟶ □(isVal<thd3<$ic_res>>
∧ @thd3<ic_res> = #True ∧ @ten11<safer_inp> = #ROT ∧ isVal<fst3<$ic_res>>
⟶ @fst3<ic_res> = (@sev11<safer_inp>,#ZERO,#ZERO))
lemma `Prop2_2_IC_4:` ⊢ program ⟶ □(isVal<thd3<$ic_res>>
∧ @thd3<ic_res> = #True ∧ @ten11<safer_inp> = #TRAN ∧ isVal<fst3<$ic_res>>
⟶ @fst3<ic_res> = (@sev11<safer_inp>,@eig11<safer_inp>,@six11<safer_inp>))
lemma `Prop2_2_IC_5:` ⊢ program ⟶ □(isVal<$w28> ∧ @w28 = #True
∧ @ten11<safer_inp> = #ROT ∧ isVal<$w27>
⟶ @w27 = (@sev11<safer_inp>,#ZERO,#ZERO))
lemma `Prop2_2_IC_6:` ⊢ program ⟶ □(isVal<$w28> ∧ @w28 = #True
∧ @ten11<safer_inp> = #TRAN ∧ isVal<$w27>
⟶ @w27 = (@sev11<safer_inp>,@eig11<safer_inp>,@six11<safer_inp>))
lemma `Prop2_2_PTC1:` ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
⟶ (isVal<snd3<$ptc_res>> ∧ @snd3<ptc_res> ≠ #ZERO
⟶ @sev11<safer_inp> = #ZERO))
lemma `Prop2_2_PTC2:` ⊢ program ⟶ □(¬(@ten11<safer_inp> = #ROT)
⟶ (isVal<snd3<$ptc_res>> ∧ @snd3<ptc_res> ≠ #ZERO

⟶ @sev11<safer_inp> = #ZERO))

lemma Prop2_2_PTC3: ⊢ program ⟶ □(isVal<snd3<$ptc_res>>
∧ @snd3<ptc_res> ≠ #ZERO ⟶ @sev11<safer_inp> = #ZERO)

lemma Prop2_2_PTC4: ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
⟶ (isVal<thd3<$ptc_res>> ∧ @thd3<ptc_res> ≠ #ZERO
⟶ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO))

lemma Prop2_2_PTC5: ⊢ program ⟶ □(¬(@ten11<safer_inp> = #ROT)
⟶ (isVal<thd3<$ptc_res>> ∧ @thd3<ptc_res> ≠ #ZERO
⟶ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO))

lemma Prop2_2_PTC6: ⊢ program ⟶ □(isVal<thd3<$ptc_res>>
∧ @thd3<ptc_res> ≠ #ZERO
⟶ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO)

theorem Prop2_2_main1: ⊢ program ⟶ □(isVal<$w30>
∧ @w30 ≠ #ZERO ⟶ @sev11<safer_inp> = #ZERO)

theorem Prop2_2_main2: ⊢ program ⟶ □(isVal<$w31> ∧ @w31 ≠ #ZERO
⟶ @sev11<safer_inp> = #ZERO ∧ @eig11<safer_inp> = #ZERO)

Requirement 3

lemma Prop3_GC_1: ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN
∧ @nin11<safer_inp> ≠ #ZERO ⟶ isVal<snd3<$gc_res>>
⟶ @snd3<gc_res> ≠ #(ZERO,ZERO,ZERO))

lemma Prop3_GC_2: ⊢ program ⟶ □(@ten11<safer_inp> = #ROT ∧
(@six11<safer_inp>,@nin11<safer_inp>,@eig11<safer_inp>) ≠ #(ZERO,ZERO,ZERO)
⟶ isVal<snd3<$gc_res>> ⟶ @snd3<gc_res> ≠ #(ZERO,ZERO,ZERO))

lemma Prop3_GC_3: ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN
∧ @nin11<safer_inp> ≠ #ZERO ⟶ isVal<$w23>
⟶ @w23 ≠ #(ZERO,ZERO,ZERO))

lemma Prop3_GC_4: ⊢ program ⟶ □(@ten11<safer_inp> = #ROT ∧
(@six11<safer_inp>,@nin11<safer_inp>,@eig11<safer_inp>) ≠ #(ZERO,ZERO,ZERO)
⟶ isVal<$w23> ⟶ @w23 ≠ #(ZERO,ZERO,ZERO))

lemma Prop3_IC_1: ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN
∧ @nin11<safer_inp> ≠ #ZERO
⟶ isVal<fst3<$ic_res>> ⟶ @fst3<ic_res> = #(ZERO,ZERO,ZERO))

lemma Prop3_IC_2: ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
∧ @six11<safer_inp>,@nin11<safer_inp>,@eig11<safer_inp>) ≠ #(ZERO,ZERO,ZERO)
⟶ isVal<fst3<$ic_res>> ⟶ @fst3<ic_res> = #(ZERO,ZERO,ZERO)

theorem Prop3_main1: ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN
∧ @nin11<safer_inp> ≠ #ZERO
⟶ isVal<$w27> ⟶ @w27 = #(ZERO,ZERO,ZERO))
lemma Prop3_main2: ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
∧ (@six11<safer_inp>,@nin11<safer_inp>,@eig11<safer_inp> ≠ #(ZERO,ZERO,ZERO)
⟶ isVal<$w27> ⟶ @w27 = #(ZERO,ZERO,ZERO))
lemma Prop3_2_GC_1: ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN ∧
@nin11<safer_inp> = #ZERO ⟶ (isVal<snd3<$gc_res>>
⟶ @snd3<gc_res> = #(ZERO,ZERO,ZERO)) ∧ (isVal<fst3<$gc_res>>
⟶ @fst3<gc_res> = (@sev11<safer_inp>,@eig11<safer_inp>,@six11<safer_inp>)))
lemma Prop3_2_GC_2: ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
∧ (@six11<safer_inp>,@nin11<safer_inp>,@eig11<safer_inp>) = #(ZERO,ZERO,ZERO)
⟶ (isVal<snd3<$gc_res>> ⟶ @snd3<gc_res> = #(ZERO,ZERO,ZERO))
∧ (isVal<fst3<$gc_res>> ⟶ @fst3<gc_res> = (@sev11<safer_inp>,#ZERO,#ZERO)))
lemma Prop3_2_GC_3: ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN
∧ @nin11<safer_inp> = #ZERO ⟶ (isVal<$w23>
∧ ⟶ @w23 = #(ZERO,ZERO,ZERO)) (isVal<$w22>
⟶ @w22 = (@sev11<safer_inp>,@eig11<safer_inp>,@six11<safer_inp>)))
lemma Prop3_2_GC_4: ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
∧ (@six11<safer_inp>,@nin11<safer_inp>,@eig11<safer_inp>)
= #(ZERO,ZERO,ZERO) ⟶ (isVal<$w23>
⟶ @w23 = #(ZERO,ZERO,ZERO))
∧ (isVal<$w22> ⟶ @w22 = (@sev11<safer_inp>,#ZERO,#ZERO)))
lemma Prop3_2_IC_1: ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN
∧ @nin11<safer_inp> = #ZERO ⟶ isVal<fst3<$ic_res>> ⟶ @fst3<ic_res>
= (@sev11<safer_inp>,@eig11<safer_inp>,@six11<safer_inp>))
lemma Prop3_2_IC_2: ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
∧ (@six11<safer_inp>,@nin11<safer_inp>,@eig11<safer_inp>)
= #(ZERO,ZERO,ZERO) ⟶ isVal<fst3<$ic_res>>
⟶ @fst3<ic_res> = (@sev11<safer_inp>,#ZERO,#ZERO))
theorem Prop3_main3: ⊢ program ⟶ □(@ten11<safer_inp> = #TRAN
∧ @nin11<safer_inp> = #ZERO)) ⟶ isVal<$w27>
⟶ @w27 = (@sev11<safer_inp>,@eig11<safer_inp>,@six11<safer_inp>)
theorem Prop3_main4: ⊢ program ⟶ □(@ten11<safer_inp> = #ROT
∧ (@six11<safer_inp>,@nin11<safer_inp>,@eig11<safer_inp>)
= #(ZERO,ZERO,ZERO)) ⟶ isVal<$w27>
⟶ @w27 = (@sev11<safer_inp>,#ZERO,#ZERO)

Requirement 4

lemma `lrud_bf1:` LRUD_Type x ∨ BF_Type x
lemma `lrud_bf2:` (LRUD_Type x) ≠ (BF_Type x)
lemma `lrud_bf3:` (LRUD_Type x) = (¬BF_Type x)
lemma `lrud_bf4:` (BF_Type x) = (¬LRUD_Type x)
lemma `lrud_bf_types1:` xs ≠ [] ⟶ (LRUD_Types xs) ⟶ (¬BF_Types xs)
lemma `lrud_bf_types2:` xs ≠ [] ⟶ (BF_Types xs) ⟶ (¬LRUD_Types xs)
lemma `lrud_bf_types3:` LRUD_Type x ⟶ ¬BF_Types (x#xs)
lemma `lrud_set1:` (LRUD_Types xs) =(set xs ⊆ LRUD_set)
lemma `lrud_set2:` LRUD_Type x ⟶ x ∈ LRUD_set
lemma `bf_set1:` (BF_Types xs) = (set xs ⊆ BF_set)
lemma `bf_set2:` BF_Type x ⟶ x ∈ BF_set
lemma `lrud_bf_set1:` (x ∈ BF_set) = (x ∉ LRUD_set)
lemma `lrud_bf_add:` ⟦ BF_Types bs; LRUD_Types ls; tc bs ; tc ls ⟧
⟹ tc (ls @ bs)
lemma `bf_add:` ⟦ BF_Types as; BF_Types bs ⟧ ⟹ BF_Types (as @ bs)
lemma `lrud_add:` ⟦ LRUD_Types as; LRUD_Types bs ⟧
⟹ LRUD_Types (as @ bs)
lemma `lrud_bf_add2:` ⟦ BF_Types bs; LRUD_Types ls; tc bs ; tc ls ⟧
⟹ tc (bs @ ls)
lemma `assoc4:` (a @ b @ c @ d) = ((a @ b) @ (c @ d))
lemma `Prop4_LRUD_1:` ⊢ program
⟶ □(isVal<fst<$lrud_res>> ⟶ tc <@fst<lrud_res>>)
lemma `Prop4_LRUD_2:` ⊢ program
⟶ □(isVal<snd<$lrud_res>> ⟶ tc <@snd<lrud_res>>)
lemma `Prop4_LRUD_3:` ⊢ program ⟶ □(isVal<fst<$lrud_res>>
∧ isVal<snd<$lrud_res>> ⟶ tc <@fst<lrud_res> @ @snd<lrud_res>>)
lemma `Prop4_LRUD_4:` ⊢ program ⟶ □(isVal<$w36> ⟶ tc <@w36>)
lemma `Prop4_LRUD_5:` ⊢ program ⟶ □(isVal<$w37> ⟶ tc <@w37>)
lemma `Prop4_LRUD_6a:` ⊢ program
⟶ □(isVal<fst<$lrud_res>> = isVal<snd<$lrud_res>>)
lemma `Prop4_LRUD_6:` ⊢ program
⟶ □(isVal<$w36> ∧ isVal<$w37> ⟶ tc <@w36 @ @w37>)
lemma `Prop4_LRUD_7:` ⊢ program
⟶ □(isVal<fst<$lrud_res>> ⟶ LRUD_Types <@fst<lrud_res>>)

lemma Prop4_LRUD_8: ⊢ program
⟶ □(isVal<snd<$lrud_res>> ⟶ LRUD_Types <@snd<lrud_res>>)
lemma Prop4_LRUD_9: ⊢ program ⟶ □(isVal<fst<$lrud_res>>
∧ isVal<snd<$lrud_res>> ⟶ LRUD_Types <@fst<lrud_res> @ @snd<lrud_res>>)
lemma Prop4_LRUD_10: ⊢ program
⟶ □(isVal<$w36> ⟶ LRUD_Types <@w36>)
lemma Prop4_LRUD_11: ⊢ program
⟶ □(isVal<$w37> ⟶ LRUD_Types <@w37>)
lemma Prop4_LRUD_12a: ⊢ program
⟶ □(isVal<fst<$lrud_res>> = isVal<snd<$lrud_res>>)
lemma Prop4_LRUD_12: ⊢ program
⟶ □(isVal<$w36> ∧ isVal<$w37> ⟶ LRUD_Types <@w36 @ @w37>)
lemma Prop4_BF_1: ⊢ program ⟶ □(isVal<fst<$bf_res>> ⟶ tc <@fst<bf_res>>)
lemma Prop4_BF_2: ⊢ program ⟶ □(isVal<snd<$bf_res>> ⟶ tc <@snd<bf_res>>)
lemma Prop4_BF_3: ⊢ program ⟶ □(isVal<fst<$bf_res>> ∧ isVal<snd<$bf_res>>
⟶ tc <@fst<bf_res> @ @snd<bf_res>>)
lemma Prop4_BF_4: ⊢ program ⟶ □(isVal<$w38> ⟶ tc <@w38>)
lemma Prop4_BF_5: ⊢ program ⟶ □(isVal<$w39> ⟶ tc <@w39>)
lemma Prop4_BF_6a: ⊢ program
⟶ □(isVal<fst<$bf_res>> = isVal<snd<$bf_res>>)
lemma Prop4_BF_6: ⊢ program
⟶ □(isVal<$w38> ∧ isVal<$w39> ⟶ tc <@w38 @ @w39>)
lemma Prop4_BF_7: ⊢ program
⟶ □(isVal<fst<$bf_res>> ⟶ BF_Types <@fst<bf_res>>)
lemma Prop4_BF_8: ⊢ program
⟶ □(isVal<snd<$bf_res>> ⟶ BF_Types <@snd<bf_res>>)
lemma Prop4_BF_9: ⊢ program ⟶ □(isVal<fst<$bf_res>>
∧ isVal<snd<$bf_res>> ⟶ BF_Types <@fst<bf_res> @ @snd<bf_res>>)
lemma Prop4_BF_10: ⊢ program ⟶ □(isVal<$w38> ⟶ BF_Types <@w38>)
lemma Prop4_BF_11: ⊢ program ⟶ □(isVal<$w39> ⟶ BF_Types <@w39>)
lemma Prop4_BF_12a: ⊢ program ⟶ □(isVal<fst<$bf_res>> = isVal<snd<$bf_res>>)
lemma Prop4_BF_12: ⊢ program
⟶ □(isVal<$w38> ∧ isVal<$w39> ⟶ BF_Types <@w38 @ @w39>)
lemma Prop4_TJ_l1: ⊢ program ⟶ □(isVal<$tj_res> ⟶ tc<@tj_res>)
lemma Prop4_TJ_l2: ⊢ program ⟶ □(isVal<$w41> ⟶ tc<@w41>)
theorem Prop4_main: ⊢ program
⟶ □(isVal<six6<$safer_res>> ⟶ tc <@six6<safer_res>>)

A.6 Expression layer integration

theorem `vdmexe`: $[\![(v,hh,p) = exe\ E\ h\ e;\ \{\} \rhd e : A]\!] \Longrightarrow A\ E\ h\ hh\ v\ p$

A.6.1 The `even-odd` example

theorem `body1`: $\{\} \rhd body : \lambda\ E\ h\ hh\ v\ p.\ \forall\ i\ r.\ E\ x = (I\ i) \longrightarrow v = (I\ (i+1))$

lemma `vdmexe2`: $[\![\{\} \rhd e : P;\ (v,hh,p) = exe\ E\ h\ e]\!] \Longrightarrow P\ E\ h\ hh\ v\ p$

lemma `vdmexe3`:

$\forall\ E\ h\ hh\ v\ p\ e\ P.\ (\{\} \rhd e : P) \wedge ((v,hh,p) = exe\ E\ h\ e) \longrightarrow P\ E\ h\ hh\ v\ p$

lemma `exebody1`: $(vv,hh,st) = exebody\ (I\ n)\ h \Longrightarrow vv = I\ (n+1)$

lemma `l1`: $\vdash program \longrightarrow \Box((\exists\ n.\ \$w1 = \#(I\ n)) \vee \$w1 = \#\bot)$
$\wedge\ (\exists\ n.\ \$w2 = \#(I\ n)) \vee \$w2 = \#\bot)$
$\wedge\ (\exists\ n.\ \$even\_res = \#(I\ n)) \vee \$even\_res = \#\bot)$
$\wedge\ (\exists\ n.\ \$odd\_res = \#(I\ n)) \vee \$odd\_res = \#\bot))$

theorem `main`:

$\vdash program \longrightarrow \Box((\forall\ n.\ \$w1 = \#(I\ n)) \longrightarrow \#n \in \#Even)$
$\wedge\ (\forall\ n.\ \$w2 = \#(I\ n)) \longrightarrow \#n \in \#Odd)$
$\wedge\ (\forall\ n.\ \$even\_res = \#(I\ n)) \longrightarrow \#n \in \#Even)$
$\wedge\ (\forall\ n.\ \$odd\_res = \#(I\ n)) \longrightarrow \#n \in \#Odd))$

Appendix B

Scheduling proofs

This appendix details the proofs that self-output scheduling, and hierarchical scheduling without nested boxes, refine lock step scheduling. None of these proofs has been mechanised, since this require a deep embedding. The proofs follow Lamport's structured way of writing mathematical proofs [122]: the proof is written hierarchically; sub-goal numbering and indention is used to separate each level where the j^{th} sub-goal of the current level i proof is labelled $\langle i \rangle j$; $\langle i \rangle j$ Q.E.D denotes the proof of the current level $i-1$ goal; implication is written ASSUME: A PROVE: B; and finally, CASE replaces ASSUME in case analysis, while the goal remains the same as the parent goal.

B.1 Proof of self-out scheduling

Firstly, the *Selfout* state in self-out scheduling is subsumed by *Runnable* in lock-step scheduling. However, in the other direction, a *Runnable* (lock-step) state, is not necessary *Selfout*. Let the self-out states be postfixed by ^{s}. This relationship of *Runnable* and *Selfout* between lock-step and self-out scheduling, is then expressed as:

Lemma B.1. $st_i = Runnable$ *iff* $st_i \in \{Runnable^s, Selfout^s\}$

Due to the $\overline{\cdots}$ notation for refinement mapping in TLA, this ^{s} postfixing is not necessary and this is thus not used henceforth. Firstly, Theorem 4.1 is formalised as $sH \Rightarrow H$, where the self-out scheduling sH is formalised in Figure 4.4 and lock-step scheduling H is formalised in Figure 4.2. Both sH and H hides the scheduling state st and the scheduler s with $\exists$. The scheduling proof is achieved by proving the $sH^l \Rightarrow \overline{H^l}$, where $\overline{H}$ expresses $H[\overline{s}/s, \overline{st}/st]$, i.e. s and st replaced by the witnesses $\overline{s}$ and $\overline{st}$ respectively. Theorem 4.1 follows by applying rule (E1) to $\overline{H^l}$ which introduces $\exists\ s, st.H^l$, which equals H by definition. The proof is then concluded by applying rule (E2) to sH^l which introduces $\exists\ s, st.sH^l$, which equals sH by definition. Now, $\overline{s}$ is trivially defined as

$\overline{s} \triangleq s$. For *st* the witness follows from Lemma B.1, and is defined as follows for each $i \in BS$:

$$\overline{st_i} \triangleq \textbf{if } st_i = \mathit{Selfout} \textbf{ then } \mathit{Runnable} \textbf{ else } st_i$$

To prove Theorem 4.1, sH' is first strengthened by the invariant K

$$K \triangleq s = \mathit{Execute} \Rightarrow \forall\, i \in BS.\ st_i \neq \mathit{Matchfail},$$

proved in the following lemma

Lemma B.2. $I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res,st\rangle} \Rightarrow \Box K$.

To enhance readability the following lemma is first proved

Lemma B.3. $I \wedge \Box[\mathcal{S} \wedge \mathcal{N} \wedge K \wedge K']_{\langle s,ws,res,st\rangle} \Rightarrow \overline{I} \wedge \Box[\overline{\mathcal{S}} \wedge \bigwedge_{i\in BS} \overline{\mathcal{B}_i}]_{\overline{\langle s,ws,res,st\rangle}}$,

which add K and K' to the assumption and uses the refinement mapping in the conclusion. Note, that $\overline{\cdots}$ distributes over all operators used in the proof. Lemma B.2 and Lemma B.3 are proved separately in the following two section, and these results are used in the final section to prove Theorem 4.1.

B.1.1 Proof of Lemma B.2

$\langle 1\rangle 1$. ASSUME: $I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res,st\rangle}$
PROVE: $\Box K$
PROOF: By rule (INV1) and some simplification the proof reduces to:
$\langle 2\rangle 1$. ASSUME: I
PROVE: K
PROOF: Unfolding I gives $\forall\, i \in BS.\ st_i = \mathit{Runnable}$, hence $\forall\, i \in BS.\ st_i \neq \mathit{Matchfail}$, thus K. $\therefore$
$\langle 2\rangle 2$. ASSUME: $K \wedge \langle s,ws,res,st\rangle' = \langle s,ws,res,st\rangle$
PROVE: K'
PROOF: Trivial since s and the *st* are the free variables of K. $\therefore$
$\langle 2\rangle 3$. ASSUME: 1. K
2. $\mathcal{S}$
3. $\mathcal{N}$
PROVE: K'
PROOF: The proof is by case-analysis on the scheduling phase s:
$\langle 3\rangle 1$. CASE: $s = \mathit{Execute}$
PROOF: Assumption $\langle 3\rangle 1$ applied to assumption $\langle 2\rangle 3.2$ implies that $s' = \mathit{Super}$, hence $s' \neq \mathit{Execute}$ and $s' = \mathit{Execute} \Rightarrow \forall\, i \in BS.\ st'_i \neq \mathit{Matchfail}$, thus K'. $\therefore$
$\langle 3\rangle 2$. CASE: $s = \mathit{Super}$

PROOF: Assumption ⟨3⟩2 and ⟨2⟩3.2 implies that $s' = \mathit{Execute}$. Thus, $\forall\, i \in \mathit{BS}.\ st'_i \neq \mathit{Matchfail}$ must hold. By the definition of $\forall$ and rule (F2), the proof reduces to the following proof for a fixed but arbitrary $i \in \mathit{BS}$:
⟨4⟩1. PROVE: $st'_i \neq \mathit{Matchfail}$
PROOF: Assumption ⟨2⟩3.3 implies

$$\bigwedge_{i \in BS} (s = \mathit{Execute} \Rightarrow s\mathcal{B}_i^e) \wedge (s = \mathit{Super} \Rightarrow s\mathcal{B}_i^s)$$

hence

$$(s = \mathit{Execute} \Rightarrow s\mathcal{B}_i^e) \wedge (s = \mathit{Super} \Rightarrow s\mathcal{B}_i^s),$$

with assumption ⟨3⟩2 the following hold

$$s\mathcal{B}_i^s. \tag{B.1}$$

The proof follows by a case-split on Q_i:
⟨5⟩1. CASE: Q_i
PROOF: By assumption ⟨5⟩1 and (B.1) st_i is either *Runnable* or *Selfout*. Thus, $st'_i \neq \mathit{Matchfail}$. ∴.
⟨5⟩2. CASE: $\neg Q_i$
PROOF: By assumption ⟨5⟩2 and (B.1) $st_i = \mathit{Blocked}$, thus $st'_i \neq \mathit{Matchfail}$. ∴.
⟨5⟩3. Q.E.D.
PROOF: By ⟨5⟩1 and ⟨5⟩2. ∴.
⟨4⟩2. Q.E.D.
PROOF: By ⟨4⟩1. ∴.
⟨3⟩3. Q.E.D.
PROOF:By ⟨3⟩1 and ⟨3⟩2. ∴.
⟨2⟩4. Q.E.D.
PROOF:By ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3. ∴.
⟨1⟩2. Q.E.D.
PROOF: By ⟨1⟩1. ∴.

B.1.2 Proof of Lemma B.3

⟨1⟩1. ASSUME: $I \wedge \Box[\mathcal{S} \wedge \mathcal{N} \wedge K \wedge K']_{\langle s,ws,res,st\rangle}$
PROVE: $\overline{I} \wedge \Box[\overline{\mathcal{S}} \wedge \bigwedge_{i \in BS} \overline{\mathcal{B}_i}]_{\overline{\langle s,ws,res,st\rangle}}$
PROOF: A specialisation of (TLA2) using (STL4) reduces the proof to
⟨2⟩1. ASSUME: I
PROVE: $\overline{I}$
PROOF: This is trivial since simplification implies $I \equiv \overline{I}$. ∴.
⟨2⟩2. ASSUME: $\mathcal{S} \wedge \mathcal{N} \wedge K \wedge K'$
PROVE: $\overline{\mathcal{S}} \wedge \bigwedge_{i \in BS} \overline{\mathcal{B}_i}$
PROOF: Unfolding $\mathcal{N}$ and some simplification reduces the proof to:

⟨3⟩1. ASSUME: $\mathcal{S} \wedge K \wedge K'$
PROVE: $\overline{\mathcal{S}}$
PROOF: This is trivial since simplification implies $\mathcal{S} \equiv \overline{\mathcal{S}}$. ∴.
⟨3⟩2. ASSUME: 1. K
2. K'
3. $\bigwedge_{i \in BS} \big(s = \mathit{Execute} \Rightarrow s\mathcal{B}_i^e\big) \wedge (s = \mathit{Super} \Rightarrow s\mathcal{B}_i^s)$
PROVE: $\bigwedge_{i \in BS} \overline{\mathcal{B}_i}$
PROOF: It is sufficient to show the goal for an arbitrary but fixed box $i \in BS$. This becomes obvious since $\bigwedge_{i \in BS}$ could be rewritten to bounded quantification $\forall i \in BS$, and then apply rule (F1) and (F2), using the definition of $\forall$ $(\neg\exists\neg)$. Assuming that box i is fixed, the goal is thus, by applying (F1) in a backwards style, reduced to
⟨4⟩1. PROVE: $\overline{\mathcal{B}_i}$
PROOF: Firstly, assumption ⟨3⟩2.3 is reduced to the i conjunct:

$$\big(s = \mathit{Execute} \Rightarrow s\mathcal{B}_i^e\big) \wedge (s = \mathit{Super} \Rightarrow s\mathcal{B}_i^s) \tag{B.2}$$

where i is the same fixed identifier as in the current goal. The proof follows by a case split on s:
⟨5⟩1. CASE: $s = \mathit{Execute}$
PROOF: Assumption ⟨5⟩1 and (B.2) implies

$$s\mathcal{B}_i^e \tag{B.3}$$

Further, assumption ⟨5⟩1 implies that $\overline{st_i} = \mathit{Execute}$. By standard equational reasoning it is trivial to show that under this assumption

$$\overline{\mathcal{B}_i} \equiv \overline{\mathcal{B}_i^e}$$

holds. Thus, it is sufficient to show $\overline{\mathcal{B}_i^e}$. The proof then follows by case analysis on the SO property:
⟨6⟩1. CASE: SO
PROOF: The proof follows by case analysis on st_i:
⟨7⟩1. CASE: $st_i = \mathit{Runnable}$
PROOF: By assumption ⟨6⟩1, the definition of SO and assumption ⟨7⟩1 a contradiction is obtained. ∴.
⟨7⟩2. CASE: $st_i = \mathit{Matchfail}$
PROOF: By assumption ⟨5⟩1, ⟨7⟩2 and ⟨3⟩2, a contradiction is obtained. ∴.
⟨7⟩3. CASE: $st_i = \mathit{Blocked}$
PROOF: By (B.3), assumption ⟨5⟩1, ⟨6⟩1 and ⟨7⟩3, the definition of $s\mathcal{B}_i^e$ implies:

$$\langle iws_i, res_i, st_i\rangle' = \langle iws_i, res_i, st_i\rangle$$

which implies that

$$\overline{\langle iws_i, res_i, st_i\rangle}' = \overline{\langle iws_i, res_i, st_i\rangle}.$$

Further, assumption ⟨7⟩3 implies

$$\overline{st_i} = \mathit{Blocked}$$

hence $\overline{\mathcal{B}_i^e}$ holds by definition. ∴

⟨7⟩4. CASE: $st_i = \mathit{Selfout}$

PROOF: By (B.3), assumption ⟨5⟩1, ⟨6⟩1 and ⟨7⟩4, the definition of $s\mathcal{B}_i^e$ implies:

$$\langle iws_i, res_i, st_i\rangle' = \mathit{execute}(rs_i, iws_i),$$

which means that

$$\overline{\langle iws_i, res_i, st_i\rangle}' = \overline{\mathit{execute}(rs_i, iws_i)}, \tag{B.4}$$

Further, assumptions ⟨7⟩4 implies (by definition)

$$\overline{st_i} = \mathit{Runnable}.$$

which, together with (B.4), implies $\overline{\mathcal{B}_i^e}$. ∴

⟨7⟩5. Q.E.D.

PROOF: By ⟨7⟩1, ⟨7⟩2, ⟨7⟩3 and ⟨7⟩4. ∴

⟨6⟩2. CASE: $\neg SO$

PROOF: The proof follows by case analysis on st_i:

⟨7⟩1. CASE: $st_i = \mathit{Runnable}$

PROOF: (B.3), assumption ⟨5⟩1, ⟨6⟩2 and ⟨7⟩1, and the definition of $s\mathcal{B}_i^e$ implies:

$$\langle iws_i, res_i, st_i\rangle' = \mathit{execute}(rs_i, iws_i)$$

hence

$$\overline{\langle iws_i, res_i, st_i\rangle}' = \overline{\mathit{execute}(rs_i, iws_i)}. \tag{B.5}$$

Moreover, assumption ⟨7⟩1 implies that

$$\overline{st_i} = \mathit{Runnable}.$$

This and (B.5) implies $\overline{\mathcal{B}_i^e}$. ∴

⟨7⟩2. CASE: $st_i = \mathit{Matchfail}$

PROOF: Assumption ⟨5⟩1, ⟨7⟩2 and ⟨3⟩2, and the definition of K induces a contradiction. ∴

⟨7⟩3. CASE: $st_i = \mathit{Blocked}$

PROOF: (B.3), assumption ⟨5⟩1, ⟨6⟩2 and ⟨7⟩3, and the definition of $s\mathcal{B}_i^e$ implies:

$$\langle iws_i, res_i, st_i\rangle' = \langle iws_i, res_i, st_i\rangle$$

hence

$$\overline{\langle iws_i, res_i, st_i\rangle}' = \overline{\langle iws_i, res_i, st_i\rangle}.$$

Further, assumption ⟨7⟩3 implies

$$\overline{st_i} = \mathit{Blocked}$$

thus $\overline{\mathcal{B}_i^e}$ holds by definition. ∴

⟨7⟩4. CASE: $st_i = \mathit{Selfout}$

PROOF: (B.3), assumption ⟨5⟩1, ⟨6⟩2 and ⟨7⟩4, and the definition of $s\mathcal{B}_i^e$ implies

$$\langle iws_i, res_i, st_i\rangle' = \mathit{execute}(rs_i, iws_i),$$

hence

$$\overline{\langle iws_i, res_i, st_i\rangle}' = \overline{\mathit{execute}(rs_i, iws_i)}, \tag{B.6}$$

Further, assumptions ⟨7⟩4 implies (by definition)

$$\overline{st_i} = \mathit{Runnable}.$$

This and (B.6) implies $\overline{\mathcal{B}_i^e}$ by definition. ∴

⟨7⟩5. Q.E.D.

PROOF: By ⟨7⟩1, ⟨7⟩2, ⟨7⟩3 and ⟨7⟩4. ∴

⟨6⟩3. Q.E.D.

PROOF: By ⟨6⟩1 and ⟨6⟩2. ∴

⟨5⟩2. CASE: $s = \mathit{Super}$

PROOF: Assumption ⟨5⟩2 and (B.2) implies

$$s\mathcal{B}_i^s \tag{B.7}$$

Further, assumption ⟨5⟩2 implies that $\overline{s} = \mathit{Super}$. Standard equational reasoning then shows

$$\overline{\mathcal{B}_i} \equiv \overline{\mathcal{B}_i^s}$$

Thus, it is sufficient to show $\overline{\mathcal{B}_i^s}$. The proof follows by a case-split on Q_i:

⟨6⟩1. CASE: Q_i

PROOF: (B.7) and assumption ⟨6⟩1 implies

$$\begin{array}{ll} & \langle ows_i, res_i\rangle' = \langle w(iws_i), \bot\rangle \\ \wedge & st_i' = \textbf{if } \mathit{emp}(nows_i') \textbf{ then } \mathit{Selfout} \textbf{ else } \mathit{Runnable} \end{array}$$

hence

$$\begin{array}{ll} & \overline{\langle ows_i, res_i\rangle}' = \overline{\langle w(iws_i), \bot\rangle} \\ \wedge & \overline{st_i}' = \textbf{if } \overline{\mathit{emp}(nows_i')} \textbf{ then } \mathit{Runnable} \textbf{ else } \mathit{Runnable} \end{array}$$

hence

$$\overline{\langle ows_i, res_i, st_i\rangle}' = \overline{\langle w(iws_i), \bot, Runnable\rangle}$$

Thus, since assumption ⟨6⟩1 implies $\overline{Q_i}$, $\overline{\mathcal{B}_i^s}$ holds. ∴.

⟨6⟩2. CASE: $\neg Q_i$

PROOF: (B.7) and assumption ⟨6⟩2 implies

$$\langle ows_i, res_i, st_i\rangle' = \langle ows_i, res_i, Blocked\rangle,$$

hence

$$\overline{\langle ows_i, res_i, st_i\rangle}' = \overline{\langle ows_i, res_i, Blocked\rangle},$$

Thus, since assumption ⟨6⟩2 implies that $\overline{\neg Q_i}$, $\overline{\mathcal{B}_i^s}$ holds. ∴.

⟨6⟩3. Q.E.D.

PROOF: By ⟨6⟩1 and ⟨6⟩2. ∴.

⟨5⟩3. Q.E.D.

PROOF: By ⟨5⟩1 and ⟨5⟩2. ∴.

⟨4⟩2. Q.E.D.

PROOF: By ⟨4⟩1. ∴.

⟨3⟩3. Q.E.D.

PROOF: By ⟨3⟩1 and ⟨3⟩2 . ∴.

⟨2⟩3. ASSUME: $\langle s,ws,res,st\rangle' = \langle s,ws,res,st\rangle$

PROVE: $\overline{\langle s,ws,res,st\rangle}' = \overline{\langle s,ws,res,st\rangle}$

PROOF: This is trivial. ∴.

⟨2⟩4. Q.E.D.

PROOF: By ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3. ∴.

⟨1⟩2. Q.E.D.

PROOF: By ⟨1⟩1. ∴.

B.1.3 Proof of Theorem 4.1

⟨1⟩1. ASSUME: $\exists s,\ st.\ I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res,st\rangle}$

PROVE: $\exists s,\ st.\ I \wedge \Box[\mathcal{S} \wedge \bigwedge_{i\in BS} \mathcal{B}_i]_{\langle s,ws,res,st\rangle}$

PROOF: Since s and st are $\exists$-bound in the goal, rule (E2) reduces the goal to:

⟨2⟩1. ASSUME: $I \wedge \Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res,st\rangle}$

PROVE: $\exists s,res,st : I \wedge \Box[\mathcal{S} \wedge \bigwedge_{i\in BS} \mathcal{B}_i]_{\langle s,ws,res,st\rangle}$

PROOF: With the substitutions $[\overline{s}/s, \overline{st}/st]$ sequentially applied to (E1), the goal reduces to:

⟨3⟩1. PROVE: $\overline{I} \wedge \Box[\overline{\mathcal{S}} \wedge \bigwedge_{i\in BS} \overline{\mathcal{B}_i}]_{\overline{\langle s,ws,res,st\rangle}}$

PROOF: Assumption ⟨2⟩1 applied to Lemma B.2 implies $\Box K$. Rule (INV2) implies

$$\Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res,st\rangle} \equiv \Box[\mathcal{S} \wedge \mathcal{N} \wedge K \wedge K']_{\langle s,ws,res,st\rangle} \tag{B.8}$$

Moreover, assumption $\langle 2\rangle 1$ implies $\Box[\mathcal{S} \wedge \mathcal{N}]_{\langle s,ws,res,st\rangle}$. This, (B.8) and assumption $\langle 2\rangle 1$ implies

$$I \wedge \Box[\mathcal{S} \wedge \mathcal{N} \wedge K \wedge K']_{\langle s,ws,res,st\rangle}$$

which, applied to Lemma B.3 implies the goal. ∴

$\langle 3\rangle 2$. Q.E.D.

PROOF: By $\langle 3\rangle 1$. ∴

$\langle 2\rangle 2$. Q.E.D.

PROOF: By $\langle 2\rangle 1$. ∴

$\langle 1\rangle 2$. Q.E.D.

PROOF: By $\langle 1\rangle 1$. ∴

B.2 Proof of hierarchical scheduling

The proof of Theorem 6.1 requires strengthening by the following invariants:

Lemma B.4. $hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E}]_{\langle s,ws,inp,res,st,pc,expr\rangle} \Rightarrow \Box K$

Lemma B.5. $hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K']_{\langle s,ws,inp,res,st,pc,expr\rangle} \Rightarrow \Box L$

Lemma B.6. $hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K' \wedge L \wedge L']_{\langle s,ws,inp,res,st,pc,expr\rangle} \Rightarrow \Box M$

Lemma B.7. $hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K' \wedge L \wedge L' \wedge M \wedge M']_{\langle s,ws,inp,res,st,pc,expr\rangle} \Rightarrow \Box N$

where K, L, M and N are defined as follows:

$$\begin{array}{lcl} K & \triangleq & s \in \{\mathit{Execute}, \mathit{Super}\} \\ L & \triangleq & \forall i \in BS.\ st_i \neq \mathit{Super} \\ M & \triangleq & s = \mathit{Execute} \Rightarrow \forall p \in BS_1.\ pc_1 \preceq p \Rightarrow st_p \in \{\mathit{Runnable}, \mathit{Blocked}, \mathit{Execute}\} \\ N & \triangleq & s = \mathit{Execute} \Rightarrow \forall p \in BS_1.\ pc_1 \prec p \Rightarrow st_p \in \{\mathit{Runnable}, \mathit{Blocked}\}. \end{array}$$

The proofs of these four lemmas are shown in the following four sections. Moreover, to prove Theorem 6.1 the refinement mapping for the ∃-bound variables (*st,s,pc,con* and *expr*) must be defined. Now, firstly *con* does not exists in H_{hi}, *pc* is a tuple of variables and not a variable, while there are several new potential box states. The remaining variables are the same, creating the following refinement mapping:

$$\begin{array}{lcl} \overline{st_i} & \triangleq & \textbf{if } st_i \in \{\mathit{Terminated}, \mathit{Execute}\} \textbf{ then } \mathit{Runnable} \textbf{ else } st_i \\ \overline{con} & \triangleq & pc_1 = env \vee st_{pc_1} \neq \mathit{Execute} \\ \overline{F} & \triangleq & [\overline{st}/st, s/s, pc_1/pc, \overline{con}/con, expr/expr]F \end{array}$$

As in the self-out scheduling proof, the following lemma is first proved, and then used in the proof of Theorem 6.1, to enhance readability:

Lemma B.8. *If* $\bigwedge_{i \in BS} flat(i)$ *and* $BS_1 = BS$ *then*

$$hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K' \wedge L \wedge L' \wedge M \wedge M' \wedge N \wedge N']_{\langle s,ws,inp,res,st,pc,expr\rangle}$$
$$\Rightarrow$$
$$\overline{I^2} \wedge \Box[\overline{\mathcal{N}^2_{seq}} \wedge \overline{\mathcal{S}^2_{seq}} \wedge \overline{\mathcal{E}}]_{\overline{\langle s,ws,inp,res,st,pc,con,expr\rangle}}$$

B.2.1 Proof of Lemma B.4

⟨1⟩1. ASSUME: $hI \wedge \Box[\, h\mathcal{N} \wedge h\mathcal{E}]_{\langle s,ws,inp,res,st,pc,expr\rangle}$
PROVE: $\Box(s \in \{Execute, Super\})$
PROOF: By rule (INV1) and some simplification the proof reduces to:
⟨2⟩1. ASSUME: hI
PROVE: $s \in \{Execute, Super\}$
PROOF: Unfolding hI implies $s = Execute$, thus the goal holds. ∴
⟨2⟩2. ASSUME: $s \in \{Execute, Super\} \wedge$
$\langle s,ws,inp,res,st,pc,expr\rangle' = \langle s,ws,inp,res,st,pc,expr\rangle$
PROVE: $s' \in \{Execute, Super\}$
PROOF: The goals follows from the first assumption conjunction $s \in \{Execute, Super\}$, and the second assumption conjunct which implies $s' = s$. ∴
⟨2⟩3. ASSUME: 1. $h\mathcal{N}$
2. $s \in \{Execute, Super\}$
PROVE: $s' \in \{Execute, Super\}$
PROOF: The proof is by case-split on the scheduling phase s. By assumption ⟨2⟩3.4 this reduces to the two cases $s = Execute$ and $s = Super$:
⟨3⟩1. CASE: $s = Execute$
PROOF: Assumption ⟨2⟩3.1 and ⟨3⟩1, with $h\mathcal{N}$ implies

$$s' = S(Execute, pc_1, T(st'_{pc}), BS_1 \cup \{env\}).$$

By the definition of S, regardless of all but the first argument, this can only result in the two cases $s' = Super$ and $s' = Execute$. ∴
⟨3⟩2. CASE: $s = Super$
PROOF: Assumptions ⟨2⟩3.1 and⟨3⟩2, and $h\mathcal{N}$ implies

$$s' = S(Super, pc_1, False, BS_1 \cup \{env\}).$$

By the definition of S this implies that $s' = Execute$. ∴
⟨3⟩3. Q.E.D.
PROOF:By ⟨3⟩1 and ⟨3⟩2. ∴
⟨2⟩4. Q.E.D.
PROOF:By ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3. ∴
⟨1⟩2. Q.E.D.
PROOF: By ⟨1⟩1. ∴

B.2.2 Proof of Lemma B.5

⟨1⟩1. ASSUME: 1. $\bigwedge_{i \in BS} flat(i)$ and $BS_1 = BS$
2. $hI \wedge \Box[\, h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K']_{\langle s,ws,inp,res,st,pc,expr\rangle}$
PROVE: $\Box(\forall i \in BS.\ st_i \neq Super)$
PROOF: By rule (INV1) and some simplification the proof reduces to:
⟨2⟩1. ASSUME: hI
PROVE: $\forall i \in BS.\ st_i \neq Super$
PROOF: Unfolding of hI implies $\bigwedge_{i \in BS} st_i = Runnable$, thus the goal holds. ∴
⟨2⟩2. ASSUME: $\forall i \in BS.\ st_i \neq Super \wedge$
$\langle s,ws,inp,res,st,pc,expr\rangle' = \langle s,ws,inp,res,st,pc,expr\rangle$
PROVE: $\forall i \in BS.\ st'_i \neq Super$
PROOF: Assumption ⟨2⟩2 implies $st' = st$. Since st is a tuple and st_i is a projection of the tuple, this implies $\forall i \in BS. st'_i = st_i$. Since the assumption also implies $\forall i \in BS.\ st_i \neq Super$ the goal trivially holds. ∴
⟨2⟩3. ASSUME: 1. $h\mathcal{N}$
2. $h\mathcal{E}$
3. $K \wedge K'$
4. $\forall i \in BS.\ st_i \neq Super$
PROVE: $\forall i \in BS.\ st'_i \neq Super$
PROOF: By (F1), it is sufficient show this for an arbitrary, but fixed $i \in BS$:
⟨3⟩1. PROVE: $st'_i \neq Super$
PROOF: The proof is by case-analysis on the scheduling phase s. Assumption ⟨2⟩3.3 implies $s \in \{Execute, Super\}$, thus reducing the split to these two cases:
⟨4⟩1. CASE: $s = Execute$
PROOF: Assumption ⟨2⟩3.1 and unfolding of $h\mathcal{N}$ implies that $h\mathcal{B}_{pc} \wedge \bigwedge_{j \in (BS_1 - \{pc\})} \mathcal{U}_j$. Lemma 6.1 (in Chapter 6) implies that $BS_1 = BS$. Thus, there are two cases $i = pc_1 \Rightarrow h\mathcal{B}_i$ and $i \neq pc_1 \Rightarrow \mathcal{U}_i$. The proof thus proceeds by a case-split on $i = pc_1$:
⟨5⟩1. CASE: $i = pc$
PROOF: Assumption ⟨5⟩1 thus implies $h\mathcal{B}_i$. With this, assumption ⟨4⟩1 and Lemma 6.1 implies that

$$h\mathcal{B}_i^{ef}. \tag{B.9}$$

The proof follows by case-analysis on st_i:
⟨6⟩1. CASE: $st_i = Runnable$
PROOF: (B.9) and assumption ⟨5⟩1 implies that the st'_i value is set by $hexecute_f^{con}$. This uses $execute^{con}$, which is defined for flat Hume. Thus, it may only return *Runnable*, *Blocked* or *Matchfail*, all implying $st'_i \neq Super$. However, in the case of *Runnable*, $hexecute_f^{con}$ updates this to *Execute*, thus $st'_i \neq Super$. ∴
⟨6⟩2. CASE: $st_i = Execute$
PROOF: (B.9) and assumption ⟨6⟩2 implies that the $st'_i = Terminated$. ∴∴
⟨6⟩3. CASE: $st_i = Super$

PROOF: By contradiction between assumption ⟨2⟩3.4 and ⟨6⟩3. ∴
⟨6⟩4. CASE: $st_i \in \{Blocked, Matchfail, Terminated\}$
PROOF: (B.9) and assumption ⟨6⟩2 implies that the $st'_i = st_i$. Thus, the goal holds by assumption ⟨2⟩3.4. ∴
⟨6⟩5. Q.E.D.
PROOF: By ⟨5⟩1,⟨6⟩2,⟨6⟩3 and ⟨6⟩4. ∴
⟨5⟩2. CASE: $i \neq pc$
Assumption ⟨5⟩2 implies $\mathcal{U}_i$, which implies $st'_i = st_i$. Thus, the goal holds by assumption ⟨2⟩3.4. ∴
⟨5⟩3. Q.E.D.
PROOF: By ⟨5⟩1 and ⟨5⟩2. ∴
⟨4⟩2. CASE: $s = Super$
PROOF: Assumption ⟨2⟩3.1 and ⟨4⟩2 implies $h\mathcal{B}_i^s$ which implies $\mathcal{B}_i^{s^2}$. The proof follows by a case-split on $ao(res_i, ows_i)$:
⟨5⟩1. CASE: $ao(res_i, ows_i)$
PROOF: Assumption ⟨5⟩1 implies that $st'_i = Runnable$, thus $st'_i \neq Super$. ∴
⟨5⟩2. CASE: $\neg ao(res_i, ows_i)$
PROOF: Assumption ⟨5⟩2 implies that $st_i = Blocked$, thus $st'_i \neq Super$. ∴
⟨5⟩3. Q.E.D.
PROOF: By ⟨5⟩1 and ⟨5⟩2. ∴
⟨4⟩3. Q.E.D.
PROOF:By ⟨4⟩1 and ⟨4⟩2. ∴
⟨3⟩2. Q.E.D.
PROOF:By ⟨3⟩1. ∴
⟨2⟩4. Q.E.D.
PROOF:By ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3. ∴
⟨1⟩2. Q.E.D.
PROOF: By ⟨1⟩1. ∴

B.2.3 Proof of Lemma B.6

⟨1⟩1. ASSUME: 1. $\bigwedge_{i \in BS} flat(i)$ and $BS_1 = BS$
2. $hI \wedge \Box[\, h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K' \wedge L \wedge L']_{\langle s,ws,inp,res,st,pc,expr\rangle}$
PROVE: $\Box M$
PROOF: By rule (INV1) and some simplification the proof reduces to:
⟨2⟩1. ASSUME: hI
PROVE: M
PROOF: Unfolding of hI implies $\bigwedge_{i \in BS} st_i = Runnable$, thus $\forall i \in BS_1.st_i \in \{Runnable, Blocked, Execute\}$. ∴
⟨2⟩2. ASSUME: $M \wedge \langle s,ws,inp,res,st,pc,expr\rangle' = \langle s,ws,inp,res,st,pc,expr\rangle$
PROVE: M'
PROOF: Assumption ⟨2⟩2 implies $st' = st$. Since st is a tuple and st_i is a projection of the tuple, this implies $\forall i \in BS.st'_i = st_i$. Moreover, the assumption implies that $s' = s$ and $pc'_1 = pc_1$. Since the assumption also implies M, then M' trivially holds since all variables in it is unchanged. ∴

⟨2⟩3. ASSUME: 1. $h\mathcal{N}$
2. $h\mathcal{E}$
3. $K \wedge K'$
4. $L \wedge L'$
5. M
PROVE: M'

PROOF: The proof is by case-split on the scheduling phase s. By assumption ⟨2⟩3.4 this reduces to the two cases $s = \mathit{Execute}$ and $s = \mathit{Super}$:

⟨3⟩1. CASE: $s = \mathit{Execute}$

PROOF: Assumption ⟨2⟩3.1 and ⟨3⟩1, with $h\mathcal{N}$ implies $\bigwedge_{i\in(BS_1-\{pc\})} \mathcal{U}_i$, which implies $\bigwedge_{i\in(BS_1-\{pc\})} st'_i = st_i$. Since the update of pc_1 is monotone (based on $\prec$), the case that $\forall p \in BS_1.\ pc_1 \prec p \Rightarrow st'_p \in \{\mathit{Runnable}, \mathit{Blocked}, \mathit{Execute}\}$ is trivial. This reduces the goal to

⟨4⟩1. PROVE: $s = \mathit{Execute} \Rightarrow st'_{pc_1} \in \{\mathit{Runnable}, \mathit{Blocked}, \mathit{Execute}\}$

PROOF: First, note that since $\mathit{env} \notin BS_1$, it is assumed that $pc_1 \neq \mathit{env}$. Hence $pc_1 = \mathit{env} \vee T(pc'_1)$ is trivially reduced to $T(pc'_1)$. From assumptions ⟨2⟩3.1 and ⟨3⟩1, Lemma 6.1, and the definition of $h\mathcal{N}$, $h\mathcal{B}_i^{ef}$ holds. The proof follows by case-analysis on st_{pc_1}:

⟨5⟩1. CASE: $st_{pc_1} = \mathit{Runnable}$

Assumption ⟨5⟩1 implies that st_{pc_1} is updated by $\mathit{execute}_f^{con}$. It is trivial to show that this implies $st'_{pc_1} = \mathit{Matchfail}$ or $st'_{pc_1} = \mathit{Execute}$. The latter case trivially implies the goal since $T(pc'_1)$ fails, meaning that following $h\mathcal{N}$, $pc'_1 = pc_1$. The former case implies $T(pc'_1)$ holds. Thus, by $h\mathcal{N}$, $pc'_1 = \mathit{next\_box}(pc, BS_1 \cup \{\mathit{env}\})$. The proof follows by case analysis on $pc_1 = \mathit{first\_box}(BS_1 \cup \{\mathit{env}\})$:

⟨6⟩1. CASE: $pc_1 = \mathit{first\_box}(BS_1 \cup \{\mathit{env}\})$

PROOF: Assumptions ⟨6⟩1, ⟨2⟩3.1 and ⟨3⟩1, together with $T(pc'_1)$ implies $s' = S(\mathit{Execute}, pc'_1, \mathit{True}, BS_1 \cup \{\mathit{env}\})$ which by S implies $s' = \mathit{Super}$. Thus, the goal holds. ∴

⟨6⟩2. CASE: $pc_1 \neq \mathit{first\_box}(BS_1 \cup \{\mathit{env}\})$

PROOF: Assumption ⟨6⟩1 implies that pc'_1 is updated according to $\prec$, pc_1 and $BS_1 \cup \{\mathit{env}\})$. Since assumption ⟨2⟩3.1 and ⟨3⟩1, with $h\mathcal{N}$ implies $\bigwedge_{i\in(BS_1\setminus\{pc\})} \mathcal{U}_i$, which implies $\bigwedge_{i\in(BS_1\setminus\{pc\})} st'_i = st_i$, and $\prec$ is monotone, the goal is trivial since assumption ⟨2⟩3.5 implies M. ∴

⟨6⟩3. Q.E.D.

PROOF: By ⟨6⟩1 and ⟨6⟩2. ∴

⟨5⟩2. CASE: $st_{pc_1} = \mathit{Execute}$

Assumption ⟨5⟩2, and $h\mathcal{B}_i^{ef}$ imples $st'_{pc_1} = \mathit{Terminated}$, which implies $T(st'_{pc_1})$. The proof is similar to the $st'_{pc_1} = \mathit{Matchfail}$ case of ⟨5⟩1. ∴

⟨5⟩3. CASE: $st_{pc_1} = \mathit{Blocked}$

PROOF: Assumption ⟨5⟩3, and $h\mathcal{B}_i^{ef}$ imples $st'_{pc_1} = \mathit{Blocked}$. The proof is similar to the $st'_{pc_1} = \mathit{Matchfail}$ case of ⟨5⟩1. ∴

⟨5⟩4. CASE: $st_{pc_1} = \mathit{Super}$

PROOF: By contradiction between assumptions ⟨5⟩4 and ⟨2⟩3.4. ∴

⟨5⟩5. CASE: $st_{pc_1} \in \{\mathit{Matchfail}, \mathit{Terminated}\}$

PROOF: By contradiction between assumptions ⟨5⟩5 and ⟨2⟩3.5. ∴

⟨5⟩6. Q.E.D.
PROOF: By ⟨5⟩1,⟨5⟩2,⟨5⟩3,⟨5⟩4 and ⟨5⟩5. ∴
⟨4⟩2. Q.E.D.
PROOF:By ⟨4⟩1. ∴
⟨3⟩2. CASE: $s = Super$
PROOF: Assumptions ⟨2⟩3.1 and⟨3⟩2, and $h\mathcal{N}$ implies $s' = S(Super, pc_1, False, BS_1 \cup \{env\})$, and S then implies that $s' = Execute$. Moreover, these assumptions implies that $pc'_1 = first\_box(BS_1 \cup \{env\})$, meaning that the following must be shown.
⟨4⟩1. PROVE: $\forall p \in BS_1.st'_p \in \{Runnable, Blocked, Execute\}$
By (F1) it is sufficient to show this for an arbitrary, but fixed $i \in BS_1$:
⟨5⟩1. PROVE: $st'_i \in \{Runnable, Blocked, Execute\}$
PROOF: Assumption ⟨2⟩3.1 and ⟨3⟩2 implies $h\mathcal{B}^s_i$ which implies $\mathcal{B}^{s^2}_i$. The proof follows by a case-split on $ao(res_i, ows_i)$:
⟨6⟩1. CASE: $ao(res_i, ows_i)$
PROOF: Assumption ⟨6⟩1 implies that $st'_i = Runnable$, thus $st'_i \in \{Runnable, Blocked, Execute\}$. ∴
⟨6⟩2. CASE: $\neg ao(res_i, ows_i)$
PROOF: Assumption ⟨6⟩2 implies that $st_i = Blocked$, thus $st'_i \in \{Runnable, Blocked, Execute\}$. ∴
⟨6⟩3. Q.E.D.
PROOF: By ⟨6⟩1 and ⟨6⟩2. ∴
⟨5⟩2. Q.E.D.
PROOF:By ⟨5⟩1. ∴
⟨4⟩2. Q.E.D.
PROOF:By ⟨4⟩1. ∴
⟨3⟩3. Q.E.D.
PROOF:By ⟨3⟩1 and ⟨3⟩2. ∴
⟨2⟩4. Q.E.D.
PROOF:By ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3. ∴
⟨1⟩2. Q.E.D.
PROOF: By ⟨1⟩1. ∴

B.2.4 Proof of Lemma B.7

⟨1⟩1. ASSUME: 1. $\bigwedge_{i \in BS} flat(i)$ and $BS_1 = BS$
2. $hI \wedge \Box[\, h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K' \wedge L \wedge L' \wedge M \wedge M']_{\langle s,ws,inp,res,st,pc,expr\rangle}$
PROVE: $\Box N$
PROOF: By rule (INV1) and some simplification the proof reduces to:
⟨2⟩1. ASSUME: hI
PROVE: N
PROOF: Unfolding of hI implies $\bigwedge_{i \in BS} st_i = Runnable$, thus $\forall i \in BS_1.st_i \in \{Runnable, Blocked\}$. ∴
⟨2⟩2. ASSUME: $N \wedge \langle s,ws,inp,res,st,pc,expr\rangle' = \langle s,ws,inp,res,st,pc,expr\rangle$
PROVE: N'

PROOF: Assumption $\langle 2\rangle 2$ implies $st' = st$. Since st is a tuple and st_i is a projection of the tuple, this implies $\forall i \in BS.st'_i = st_i$. Moreover, the assumption implies that $s' = s$ and $pc'_1 = pc_1$. Since the assumption also implies N, then N' trivially holds since all variables in it is unchanged. $\therefore$

$\langle 2\rangle 3$. ASSUME: 1. $h\mathcal{N}$
2. $h\mathcal{E}$
3. $K \wedge K'$
4. $L \wedge L'$
5. $M \wedge M'$
6. M

PROVE: N'

PROOF: The proof is by case-split on the scheduling phase s. By assumption $\langle 2\rangle 3.4$ this reduces to the two cases $s = Execute$ and $s = Super$:

$\langle 3\rangle 1$. CASE: $s = Execute$

PROOF: Assumption $\langle 2\rangle 3.1$ and $\langle 3\rangle 1$, with $h\mathcal{N}$ implies $\bigwedge_{i\in(BS_1-\{pc\})} \mathcal{U}_i$, which implies $\bigwedge_{i\in(BS_1-\{pc\})} st'_i = st_i$. Thus, $\forall p \in BS_1.\ pc_1 \prec p \Rightarrow st'_p \in \{Runnable, Blocked\}$. To verify M' it must be shown that $\forall p \in BS_1.\ pc'_1 \prec p \Rightarrow st'_p \in \{Runnable, Blocked\}$. Now, pc_1 is either unchanged or updated according to the monotone $\prec$ well-founded relation. Thus, $\{p \in BS_1.\ pc'_1 \prec p\} \subseteq \{p \in BS_1.\ pc_1 \prec p\}$, meaning that $\forall p \in BS_1.\ pc_1 \prec p \Rightarrow st'_p \in \{Runnable, Blocked\}$ is sufficiently strong. $\therefore$

$\langle 3\rangle 2$. CASE: $s = Super$

PROOF: Assumptions $\langle 2\rangle 3.1$ and$\langle 3\rangle 2$, and $h\mathcal{N}$ implies $s' = S(Super, pc_1, False, BS_1 \cup \{env\})$, and S then implies that $s' = Execute$. Moreover, these assumptions implies that $pc'_1 = first\_box(BS_1 \cup \{env\})$, meaning that the following must be shown.

$\langle 4\rangle 1$. PROVE: $\forall p \in BS_1.st'_p \in \{Runnable, Blocked\}$

By (F1) it is sufficient to show this for an arbitrary, but fixed $i \in BS_1$:

$\langle 5\rangle 1$. PROVE: $st'_i \in \{Runnable, Blocked\}$

PROOF: Assumption $\langle 2\rangle 3.1$ and $\langle 3\rangle 2$ implies $h\mathcal{B}^s_i$ which implies $\mathcal{B}^{s^2}_i$. The proof follows by a case-split on $ao(res_i, ows_i)$:

$\langle 6\rangle 1$. CASE: $ao(res_i, ows_i)$

PROOF: Assumption $\langle 6\rangle 1$ implies that $st'_i = Runnable$, thus $st'_i \in \{Runnable, Blocked\}$. $\therefore$

$\langle 6\rangle 2$. CASE: $\neg ao(res_i, ows_i)$

PROOF: Assumption $\langle 6\rangle 2$ implies that $st_i = Blocked$, thus $st'_i \in \{Runnable, Blocked\}$. $\therefore$

$\langle 6\rangle 3$. Q.E.D.

PROOF: By $\langle 6\rangle 1$ and $\langle 6\rangle 2$. $\therefore$

$\langle 5\rangle 2$. Q.E.D.

PROOF:By $\langle 5\rangle 1$. $\therefore$

$\langle 4\rangle 2$. Q.E.D.

PROOF:By $\langle 4\rangle 1$. $\therefore$

$\langle 3\rangle 3$. Q.E.D.

PROOF:By $\langle 3\rangle 1$ and $\langle 3\rangle 2$. $\therefore$

$\langle 2\rangle 4$. Q.E.D.

PROOF:By $\langle 2\rangle 1$, $\langle 2\rangle 2$ and $\langle 2\rangle 3$. $\therefore$
$\langle 1\rangle 2$. Q.E.D.
PROOF: By $\langle 1\rangle 1$. $\therefore$

B.2.5 Proof of Lemma B.8

$\langle 1\rangle 1$. ASSUME: 1. $\bigwedge_{i\in BS} \mathit{flat}(i)$ and $BS_1 = BS$
2. $hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K' \wedge L \wedge L' \wedge M \wedge M' \wedge N \wedge N']_{\langle s,ws,inp,res,st,pc,expr\rangle}$
PROVE: $\overline{I^2} \wedge \Box[\overline{\mathcal{N}^2_{seq}} \wedge \overline{\mathcal{S}^2_{seq}} \wedge \overline{\mathcal{E}}]_{\overline{\langle s,ws,inp,res,st,pc,con,expr\rangle}}$
PROOF: A specialisation of (TLA2) using (STL4), together with some weakening of the assumptions, reduces the proof to five cases:
$\langle 2\rangle 1$. ASSUME: hI
PROVE: $\overline{I^2}$
PROOF: $\overline{s}$, $\overline{ws}$, $\overline{st}$, $\overline{inp}$, $\overline{res}$ and $\overline{expr}$ follows directly from hI. Lemma 6.1 implies $\mathit{first\_box}(BS_1 \cup \{env\}) = \mathit{first\_box}$, thus $\overline{pc}$ follows from hI. Finally, hI implies that $s = \mathit{Execute}$ and $pc_1 = env \vee st_{pc_1} \neq \mathit{Execute}$, thus $\overline{con}$ holds. Thus, $\overline{I^2}$ holds. $\therefore$
$\langle 2\rangle 2$. ASSUME: $\overline{\langle s,ws,inp,res,st,pc,expr\rangle}' = \overline{\langle s,ws,inp,res,st,pc,expr\rangle}$
PROVE: $\overline{\langle s,ws,inp,res,st,pc,con,expr\rangle}' = \overline{\langle s,ws,inp,res,st,pc,con,expr\rangle}$
PROOF: $\overline{\langle s,ws,inp,res,st,pc,con,expr\rangle}$ is either the same variables as in $\langle s,ws,inp,res,st,pc,expr\rangle$, or defined using variables in this tuple. Thus, this trivially holds. $\therefore$
$\langle 2\rangle 3$. ASSUME: $h\mathcal{E}$
PROVE: $\overline{\mathcal{E}}$
PROOF: $h\mathcal{E}$ deviates from $\mathcal{E}$ by replacing pc with pc_1. Moreover, only wires (ws) are updated, while the guards uses s in addition to pc/pc_1. Now, since $\overline{s} = s$, $\overline{ws} = ws$ and $\overline{pc} = pc_1$ the goal follows from assumption $\langle 2\rangle 3$. $\therefore$
$\langle 2\rangle 4$. ASSUME: 1. $h\mathcal{N}$
2. $K \wedge K'$
3. $L \wedge L'$
PROVE: $\overline{\mathcal{S}_{seq^2}}$
PROOF: The proof is by case-analysis of s, which by assumption $\langle 2\rangle 4.2$ is reduced to two cases:
$\langle 3\rangle 1$. CASE: $s = \mathit{Execute}$
PROOF: Assumptions $\langle 2\rangle 4.1$ and $\langle 3\rangle 1$ implies $s' = S(\mathit{Execute}, pc'_1, pc'_1 = env \vee T(st'_{pc}), BS_1 \cup \{env\})$. The proof follows by case-analysis on $pc'_1 = \mathit{last\_box}(BS_1 \cup \{env\})$:
$\langle 4\rangle 1$. CASE: $pc'_1 = \mathit{last\_box}(BS_1 \cup \{env\})$
PROOF: The proof follows by case-analysis on $pc'_1 = env \vee T(st'_{pc})$:
$\langle 5\rangle 1$. CASE: $pc'_1 = env \vee T(st'_{pc_1})$
PROOF: With these assumption S implies that $s' = \mathit{Super}$ and consequently $\overline{s}' = \mathit{Super}$. Now, assumption $\langle 4\rangle 1$ and Lemma 6.1.1 implies that $\overline{pc} = \overline{\mathit{last\_box}}$. Moreover, since $s' = \mathit{Super}$, $\overline{con}$ holds. Thus, $\overline{\mathcal{S}_{seq^2}}$ holds. $\therefore$
$\langle 5\rangle 2$. CASE: $\neg(pc'_1 = env \vee T(st'_{pc}))$
PROOF: With these assumption S implies that $s' = \mathit{Execute}$ and consequently $\overline{s}' = \mathit{Execute}$. Assumption $\langle 4\rangle 1$ and Lemma 6.1.1 implies that

$\overline{pc} = \overline{last\_box}$. Moreover, assumption $\langle 5\rangle 1$ implies that either $pc'_1 \neq env$ and $\neg T(st'_{pc_1})$. This, together with assumption $\langle 2\rangle 4.3$ implies that

$$st'_{pc_1} \in \{Runnable, Execute\}. \tag{B.10}$$

Moreover, assumption $\langle 2\rangle 4.1$ implies $pc'_1 = pc_1$ and $h\mathcal{B}^{s^f}_{pc'_1}$, where $h\mathcal{B}^{s^f}_{pc'_1}$ has three cases: firstly, $T(st_{pc_1})$ implies $st'_{pc_1} = st_{pc_1}$ which cannot hold since it violates $\neg T(st'_{pc_1})$; secondly, $st_{pc_1} = Execute$ implies $st'_{pc_1} = Terminated$, which contradicts (B.10); finally, $st_{pc_1} = Runnable$ implies that $st'_{pc_1} \in \{Execute, Blocked, Matchfail\}$ where only $st'_{pc_1} = Execute$ does not contradict (B.10). Thus, $st'_{pc_1} = Execute$ which implies $\neg\overline{con}$. Thus, $\overline{\mathcal{S}_{seq^2}}$ implies $\overline{s}' = Super$, and $\overline{\mathcal{S}_{seq^2}}$ holds. $\therefore$

$\langle 5\rangle 3$. Q.E.D.

PROOF: By $\langle 5\rangle 1$ and $\langle 5\rangle 2$. $\therefore$

$\langle 4\rangle 2$. CASE: $pc'_1 \neq last\_box(BS_1 \cup \{env\})$

With these assumption S implies that $s' = Execute$ and consequently $\overline{s}' = Execute$. Assumption $\langle 4\rangle 1$ and Lemma 6.1.1 implies that $\overline{pc} = \overline{last\_box}$, thus $\overline{\mathcal{S}_{seq^2}}$ implies $\overline{s}' = Execute$, and $\overline{\mathcal{S}_{seq^2}}$ holds. $\therefore$

$\langle 4\rangle 3$. Q.E.D.

PROOF: By $\langle 4\rangle 1$ and $\langle 4\rangle 2$. $\therefore$

$\langle 3\rangle 2$. CASE: $s = Super$

PROOF: Assumptions $\langle 2\rangle 4.1$ and $\langle 3\rangle 2$ implies
$s' = S(Super, pc_1, False, BS_1 \cup \{env\})$, which by S implies $s' = Execute$, and thus $\overline{s}' = Execute$. Thus, $\overline{\mathcal{S}_{seq^2}}$ holds. $\therefore$

$\langle 3\rangle 3$. Q.E.D.

PROOF: By $\langle 3\rangle 1$ and $\langle 3\rangle 2$. $\therefore$

$\langle 2\rangle 5$. ASSUME: 1. $h\mathcal{N}$
2. $K \wedge K'$
3. $L \wedge L'$
4. $M \wedge M'$
5. $N \wedge N'$

PROVE: $\overline{\mathcal{N}^2_{seq}}$

PROOF: The proof is by case analysis of the scheduling state s, which by assumption $\langle 2\rangle 5.2$ is reduced to two cases:

$\langle 3\rangle 1$. CASE: $s = Execute$

PROOF: Assumption $\langle 3\rangle 1$ implies $\overline{s} = Execute$, which reduces the goal to four conjuncts, each proved separately:

$\langle 4\rangle 1$. CASE: $\overline{pc} \in BS \Rightarrow \overline{\mathcal{B}^{e^2}_{pc}}$

PROOF: Assumptions $\langle 3\rangle 1$ and $\langle 2\rangle 5$ implies $pc_1 \in BS_1 \Rightarrow h\mathcal{B}^{e}_{pc_1}$, which by assumption $\langle 1\rangle 1.1$ implies $\overline{pc} \in BS \Rightarrow h\mathcal{B}^{e^f}_{\overline{pc}}$. The case where $\overline{pc} \notin BS$ is trivial, thus $\overline{pc} \in BS$ is assumed, reducing the proof to

$\langle 5\rangle 1$. ASSUME: $h\mathcal{B}^{e^f}_{\overline{pc}}$

PROVE: $\overline{\mathcal{B}^{e^2}_{pc}}$

PROOF: The proof follows by case-analysis on st_{pc_1}:

$\langle 6\rangle 1$. CASE: $st_{pc_1} = Runnable$

PROOF: Assumption ⟨5⟩1 and ⟨6⟩1 implies that

$$\langle iws_{pc_1}, inp_{pc_1}, expr, st_{pc_1}\rangle' = execute_f^{con}(rs_{pc_1}, iws_{pc_1}, expr) \wedge res'_{pc_1} = res_{pc_1}$$

$execute_f^{con}$ only deviates from $execute^{con}$ by replacing a *Runnable* state with *Terminated*. Since $st_{pc_1} = Terminated$ implies $\overline{st_{pc_1}} = Runnable$, this implies

$$\overline{\langle iws_{pc_1}, inp_{pc_1}, expr, st_{pc_1}\rangle}' = \overline{execute^{con}(rs_{pc_1}, iws_{pc_1}, expr)} \wedge \overline{res_{pc_1}}' = \overline{res_{pc_1}}$$

All that is remaining is to show that $st_{pc_1} = Runnable$ implies that $\overline{st_{pc_1}} \neq Blocked$ and $\overline{con}$. The first conjunct is trivial. Since this is an execute step and $pc_1 \neq env$ the second conjunct reduces to show that $st_{pc_1} \neq Execute$ which is trivial. ∴.

⟨6⟩2. CASE: $st_{pc_1} = Matchfail$

PROOF: By contradiction between assumption ⟨2⟩5.4 and ⟨6⟩2. ∴.

⟨6⟩3. CASE: $st_{pc_1} = Blocked$

PROOF: Assumption ⟨6⟩3 implies $T(st_{pc_1})$, which by assumption ⟨5⟩1 and ⟨6⟩3 implies

$$\langle expr, iws_i, inp_i, res_i, st_i\rangle' = \langle expr, iws_i, inp_i, res_i, st_i\rangle$$

Assumption ⟨6⟩3 implies $\overline{st_{pc}} = Blocked$ and this implies

$$\overline{\langle expr, iws_i, inp_i, res_i, st_i\rangle}' = \overline{\langle expr, iws_i, inp_i, res_i, st_i\rangle}$$

Hence, the goal holds. ∴.

⟨6⟩4. CASE: $st_{pc_1} = Execute$

PROOF: Assumption ⟨5⟩1 and ⟨6⟩1 implies

$$\langle expr, iws_i, inp_i, res_i, st_i\rangle' = \langle expr, iws_i, inp_i, hrun(expr, inp_i), Terminated\rangle$$

which implies

$$\overline{\langle expr, iws_i, inp_i, res_i, st_i\rangle}' = \langle \overline{expr}, \overline{iws_i}, \overline{inp_i}, \overline{hrun(expr, inp_i)}, Runnable\rangle$$

Thus, all that is remaining is to show that $\overline{st_{pc}} \neq Blocked$ and $\neg\overline{con}$. The first conjunct is trivial. Since $s = Execute$ and $pc_1 \neq env$, $\neg\overline{con}$ reduces to show that $\neg(st_{pc_1} \neq Execute)$, that is $st_{pc_1} = Execute$, which holds by ⟨6⟩4. ∴.

⟨6⟩5. CASE: $st_{pc_1} = Super$

PROOF: By contradiction between assumption ⟨2⟩5.3 and ⟨6⟩5. ∴.

⟨6⟩6. CASE: $st_{pc_1} = Terminated$

PROOF: By contradiction between assumption ⟨2⟩5.4 and ⟨6⟩6. ∴.

⟨6⟩7. Q.E.D.

PROOF: By ⟨6⟩1, ⟨6⟩2, ⟨6⟩3, ⟨6⟩4, ⟨6⟩5 and ⟨6⟩6. ∴.

⟨5⟩2. Q.E.D.

PROOF: By ⟨5⟩1. ∴.

⟨4⟩2. CASE: $\bigwedge_{i \in BS-\{\overline{pc}\}} \overline{\langle iws_i, inp_i, res_i, st_i \rangle}' = \overline{\langle iws_i, inp_i, res_i, st_i \rangle}$
PROOF: Firstly, $\forall\, i \in BS_1.\, boxsch(i) = s$. Now, ⟨2⟩5.1 implies $\bigwedge_{i \in BS_1 - \{pc_1\}} .\mathcal{U}_i$ Assumtion ⟨1⟩1.1 thus implies $\bigwedge_{i \in BS-\{\overline{pc}\}} \mathcal{U}_i$. Since, $\forall\, i \in BS_1.\, boxsch(i) = s$, and assumption ⟨3⟩1, the definition of $\mathcal{U}_i$ implies $\overline{\langle iws_i, inp_i, res_i, st_i \rangle}' = \overline{\langle iws_i, inp_i, res_i, st_i \rangle}$, hence the goal holds. ∴
⟨4⟩3. CASE: $\overline{con}' = \textbf{if } \overline{pc} = env \vee \overline{st_{pc}}' \neq Runnable \textbf{ then } True \textbf{ else } \neg\overline{con}$
PROOF: The proof is by case-analysis on $pc_1 = env$:
⟨5⟩1. CASE: $pc_1 = env$
PROOF: Assumption ⟨5⟩1 implies $\overline{pc} = env$, thus it must be shown that $\overline{con}'$ holds. The proof is by case-analsys of the new scheduling state s', which by assumption ⟨2⟩5.2 results in two cases:
⟨6⟩1. CASE: $s' = Super$
PROOF: Assumptions ⟨3⟩1, ⟨6⟩1 and ⟨2⟩5.1 implies that $pc_1 = last\_box(BS_1 \cup \{env\})$. By these assumption, this implies that $pc_1' = env$, and thus $\overline{con}'$ holds. ∴
⟨6⟩2. CASE: $s' = Execute$
PROOF: Assumptions ⟨5⟩1, ⟨6⟩2, ⟨2⟩5.1 and ⟨3⟩1 implies that $pc_1 \neq last\_box(BS_1)$, since $pc_1 = env$ then implies that $s' = Super$. Thus, $pc_1' \neq pc_1$ and $pc_1 \prec pc_1'$ by the definition of *next_box* with these assumptions. Moreover, since $pc_1' \neq env$, is must be shown that $st_{pc_1'} \neq Execute$. With the above, assumption ⟨2⟩5.5 entails $st_{pc_1'} \in \{Runnable, Blocked\}$, thus $st_{pc_1'} \neq Execute$, and $\overline{con}'$ holds. ∴
⟨6⟩3. Q.E.D.
PROOF: By ⟨6⟩1 and ⟨6⟩2. ∴
⟨5⟩2. CASE: $pc_1 \neq env$
PROOF: Assumption ⟨5⟩2 implies that $\overline{pc} \neq env$. The second case that entails that $\overline{con}'$ holds iff $\overline{st_{pc}}' \neq Runnable$. By unfolding the refinement mapping, this case is implied by $st'_{pc_1} \notin \{Runnable, Execute, Terminated\}$, and the proof follows by a case split on it:
⟨6⟩1. CASE: $st'_{pc_1} \notin \{Runnable, Execute, Terminated\}$
PROOF: This requires a proof that $\overline{con}' = True$. By assumptions ⟨5⟩2 and ⟨3⟩1, and the definition of the $\overline{con}$ witness, this proof reduces to show that $st'_{pc_1} \neq Execute$, which trivially holds by assumption ⟨6⟩1. ∴
⟨6⟩2. CASE: $st'_{pc_1} \in \{Runnable, Execute, Terminated\}$
PROOF: Assumptions ⟨3⟩1, ⟨5⟩2 and ⟨6⟩2 reduces this proof to a proof of $\overline{con}' = \neg\overline{con}$. Moreover, these assumptions imply that $\overline{con}$ holds iff $st_{pc_1} \neq Execute$. The proof follows by a case-analysis:
⟨7⟩1. CASE: $st_{pc_1} = Execute$
PROOF:Assumption ⟨7⟩1 implies $\neg\overline{con}$. Thus, it must be shown that $\overline{con}'$ holds.
For $s' = Super$, assumptions ⟨3⟩1 and ⟨2⟩5.1 implies that $T(st'_{pc_1})$ holds, and $pc_1' = pc_1$. Assumption ⟨6⟩2 thus implies that $st'_{pc_1} = st'_{pc_1'} = Terminated \neq Execute$. Thus $\overline{con}'$ holds.
For $s' = Execute$, assumptions ⟨2⟩5, ⟨3⟩1.1 and ⟨7⟩1 implies $st'_{pc_1} = Terminated$, meaning that $st'_{pc_1} \neq Execute$. Thus, $T(st'_{pc_1})$ holds, and since $s' = Execute$, it must be that case that $pc_1 \prec pc_1'$ by the definition

of *next_box*. By assumption, ⟨2⟩5.5 $pc'_1 = env$ or $st'_{pc'_1} \in \{Runnable, Blocked\}$ and both cases implies $\overline{con}'$. ∴.

⟨7⟩2. CASE: $st_{pc_1} \neq Execute$

PROOF: Assumption ⟨7⟩2 implies $\overline{con}$. Thus, it must be shown that $\neg\overline{con}'$ holds. Now, assumption ⟨6⟩2 implies $st'_{pc_1} \in \{Runnable, Execute, Terminated\}$. Further, Assumption ⟨2⟩5.5 and ⟨7⟩2 implies that $st_{pc_1} \in \{Runnable, Blocked\}$. Now, with the above assumptions, assumption ⟨2⟩5.1 implies that $st_{pc_1} = Blocked$ implies that $st'_{pc_1} = Blocked$, which induces a contradiction. Hence, it must be the case that $st_{pc_1} = Runnable$. Moreover, assumption ⟨2⟩5.1 implies with this assumption, induces two possible values for st'_{pc_1}: *Matchfail* or *Execute*. Since, $st'_{pc_1} \in \{Runnable, Execute, Terminated\}$ must hold, the only valid case is $st'_{pc_1} = Execute$ which implies $\neg\overline{con}'$. ∴.

⟨7⟩3. Q.E.D.

PROOF: By ⟨7⟩1 and ⟨7⟩2. ∴.

⟨6⟩3. Q.E.D.

PROOF: By ⟨6⟩1 and ⟨6⟩2. ∴.

⟨5⟩3. Q.E.D.

PROOF: By ⟨5⟩1 and ⟨5⟩2. ∴.

⟨4⟩4. CASE: $\overline{pc}' =$
if $\overline{pc} \neq env \wedge \overline{st_{pc}}' = Runnable \Rightarrow \neg\overline{con}$
then $\overline{next\_box(pc)}$ **else** $\overline{pc}$

PROOF: Assumptions ⟨2⟩5.1 and ⟨3⟩1 implies

$$pc'_1 = \textbf{if } pc_1 = env \vee T(st'_{pc_1}) \textbf{ then } next\_box(pc_1, BS_1 \cup \{env\}) \textbf{ else } pc_1$$

Assumption ⟨1⟩1.1, Lemma 6.1.3 and the refinement mapping thus implies

$$\overline{pc}' = \textbf{if } \overline{pc} = env \vee T(st'_{pc}) \textbf{ then } \overline{next\_box}(\overline{pc}) \textbf{ else } \overline{pc}$$

Thus, it is sufficient to show that

$$(\overline{pc} = env \vee T(st'_{pc})) \equiv (\overline{pc} \neq env \wedge \overline{st_{pc}}' = Runnable \Rightarrow \neg\overline{con}),$$

which simplifies to

$$(A) \quad pc_1 = env \vee st'_{pc_1} \in \{Blocked, Matchfail, Terminated\}$$
$$\equiv$$
$$(B) \quad pc_1 \neq env \wedge st'_{pc_1} \in \{Runnable, Execute, Terminated\} \Rightarrow st_{pc_1} = Execute.$$

The $pc_1 = env$ case is trivial. The $pc_1 \neq env$ case follows by a case-analysis on st'_{pc_1}:

⟨5⟩1. CASE: $st'_{pc_1} = Runnable$

PROOF: In (A) this is *False*. For (B) is must be shown that $st_{pc_1} \neq Execute$ This is proved by contradiction. Let $st_{pc_1} = Execute$. Then assumptions ⟨2⟩5.1 and ⟨3⟩1 implies that $st'_{pc_1} = Terminated$, which contradicts assumption ⟨5⟩1. ∴.

⟨5⟩2. CASE: $st'_{pc_1} = Matchfail$
PROOF: In both (A) and (B) this is trivially *True*. ∴
⟨5⟩3. CASE: $st'_{pc_1} = Blocked$
PROOF: In both (A) and (B) this is trivially *True*. ∴
⟨5⟩4. CASE: $st'_{pc_1} = Execute$
PROOF: In (A) this is *False*. In (B) this implies $st_{pc_1} \neq Execute$, and the proof is the same as in ⟨5⟩1. ∴
⟨5⟩5. CASE: $st'_{pc_1} = Super$
PROOF: Assumption ⟨5⟩5 contradicts assumption ⟨2⟩5.2. ∴
⟨5⟩6. CASE: $st'_{pc_1} = Terminated$
PROOF: In (A) this holds. In (B) it must be shown that $st_{pc_1} = Execute$. The proof is by contradiction. Assume $st_{pc_1} \neq Execute$. By assumptions ⟨2⟩5.1 and ⟨3⟩1 it can be shown that the only case that does not contradicts ⟨5⟩6, is when $st_{pc_1} = Terminated$. However, this contradicts assumption ⟨2⟩5.4. Thus, is must be the case that $st_{pc_1} = Execute$. ∴
⟨5⟩7. Q.E.D.
PROOF: By ⟨5⟩1, ⟨5⟩2, ⟨5⟩3, ⟨5⟩4, ⟨5⟩5 and ⟨5⟩6. ∴
⟨4⟩5. Q.E.D.
PROOF: By ⟨4⟩1, ⟨4⟩2, ⟨4⟩3 and ⟨4⟩4. ∴
⟨3⟩2. CASE: $s = Super$
PROOF: Assumption ⟨3⟩2 implies $\overline{s} = Super$, which reduces the goal four conjuncts, each proved separately:
⟨4⟩1. CASE: $\bigwedge_{i \in BS} \overline{\mathcal{B}_i^{s^2}}$
PROOF: Assumptions ⟨2⟩5.1 and ⟨3⟩2 implies $\bigwedge_{i \in BS_1} h\mathcal{B}_i^s$, which by assumption ⟨1⟩1.1 implies $\bigwedge_{i \in BS} \mathcal{B}_i^{s^2}$. It is trivial to show that this implies $\forall i \in BS.\ st'_i \in \{Runnable, Blocked\}$. Moreover, $\bigwedge_{i \in BS} \mathcal{B}_i^{s^2}$ updates *ws*, *inp* and *res*, all unaffected by the refinement mapping. Thus, $\bigwedge_{i \in BS} \overline{\mathcal{B}_i^{s^2}}$ holds. ∴
⟨4⟩2. CASE: $\overline{pc}' = \overline{first\_box}$
PROOF: Assumptions ⟨2⟩5.1 and ⟨3⟩2 implies $pc'_1 = first\_box(BS_1 \cup \{env\})$. Lemma 6.1.1 then implies that $pc'_1 = first\_box$, which implies $\overline{pc} = \overline{first\_box}$ since pc_1 is the only free variable of *first_box*. ∴
⟨4⟩3. CASE: $\overline{con}' = True$
PROOF: It is trivial to show that ⟨2⟩5.1 implies $\forall i \in BS_1.\ st'_i \in \{Runnable, Blocked\}$. Following the definition of the $\overline{con}$, $s' = Super$ implies $\overline{con}$. In the case of $s' = Execute$, $pc'_1 = first\_box(BS_1 \cup \{env\})$. If $pc'_1 = env$ then $\overline{con}$ trivially holds. If $pc'_1 \neq env$ then $pc'_1 \in BS_1$. It has already been shown that $\forall i \in BS_1.\ st'_i \in \{Runnable, Blocked\}$, thus $\overline{con}$ trivially holds in this case as well. ∴
⟨4⟩4. CASE: $\overline{expr}' = \overline{expr}$
PROOF: The refinement mapping is defined as $\overline{expr} = expr$ and the assumptions ⟨2⟩5.1 and ⟨3⟩2 implies $expr' = expr$. Thus, the goal holds. ∴
⟨4⟩5. Q.E.D.
PROOF: By ⟨4⟩1, ⟨4⟩2, ⟨4⟩3 and ⟨4⟩4. ∴
⟨3⟩3. Q.E.D.
PROOF: By ⟨3⟩1 and ⟨3⟩2. ∴
⟨2⟩6. Q.E.D.

PROOF: By ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, ⟨2⟩4 and ⟨2⟩5. ∴
⟨1⟩2. Q.E.D.
PROOF: By ⟨1⟩1. ∴

B.2.6 Proof of Theorem 6.1

⟨1⟩1. ASSUME: 1. $\bigwedge_{i \in BS} flat(i)$ and $BS_1 = BS$
2. $\exists\, st,s,pc,expr.hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E}]_{\langle s,ws,inp,res,st,pc,expr\rangle}$
PROVE: $\exists\, st,s,pc,con,expr.I^2 \wedge \Box[\mathcal{N}^2_{seq} \wedge \mathcal{S}^2_{seq} \wedge \mathcal{E}]_{\langle s,ws,inp,res,st,pc,con,expr\rangle}$
PROOF: Since *st,s,pc,con* and *expr* are not free in the the goal, rule (E2) reduces the proof to:
⟨2⟩1. ASSUME: $hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E}]_{\langle s,ws,inp,res,st,pc,expr\rangle}$
PROVE: $\exists\, st,s,pc,con,expr.I^2 \wedge \Box[\mathcal{N}^2_{seq} \wedge \mathcal{S}^2_{seq} \wedge \mathcal{E}]_{\langle s,ws,inp,res,st,pc,con,expr\rangle}$
PROOF: By applying the substitutions defined by $\overline{F}$ sequentially using (E1), the goal is reduced to:
⟨3⟩1. PROVE: $\overline{I^2} \wedge \Box[\overline{\mathcal{N}^2_{seq}} \wedge \overline{\mathcal{S}^2_{seq}} \wedge \overline{\mathcal{E}}]_{\overline{\langle s,ws,inp,res,st,pc,con,expr\rangle}}$
PROOF: Applying assumptions ⟨1⟩1.1 and ⟨2⟩1 to Lemma B.4 induces $\Box K$. Rule (INV2) implies

$$\Box[h\mathcal{N} \wedge h\mathcal{E}]_{\langle s,ws,inp,res,st,pc,expr\rangle} \equiv \Box[h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K']_{\langle s,ws,inp,res,st,pc,expr\rangle}$$

The same approach can be applied to Lemma B.5, Lemma B.6 resulting in

$$\Box[h\mathcal{N} \wedge h\mathcal{E}]_{\langle s,ws,inp,res,st,pc,expr\rangle} \equiv \Box[h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K' \wedge L \wedge L' \wedge M \wedge M' \wedge N \wedge N']_{\langle s,ws,inp,res,st,pc,expr\rangle}$$

Thus, by assumption
⟨4⟩1. .
1 it can be shown that

$$hI \wedge \Box[h\mathcal{N} \wedge h\mathcal{E} \wedge K \wedge K' \wedge L \wedge L' \wedge M \wedge M' \wedge N \wedge N']_{\langle s,ws,inp,res,st,pc,expr\rangle}$$

By applying this and assumption ⟨1⟩1.1 to Lemma B.8 the goal has been proved.
∴
⟨3⟩2. Q.E.D.
PROOF: By ⟨3⟩1. ∴
⟨2⟩2. Q.E.D.
PROOF: By ⟨2⟩1. ∴
⟨1⟩2. Q.E.D.
PROOF: By ⟨1⟩1. ∴

Appendix C

SAFER program source code

```
data axis_command = NEG | ZERO | POS;
type command = (axis_command,axis_command,axis_command);
type axis_pred = (bool,bool,bool);

data thruster_name = B1 | B2 | B3 | B4 | F1 | F2 | F3 | F4 | L1R | L1F |
       R2R | R2F | L3R | L3F | R4R | R4F | D1R | D1F | D2R | D2F | U3R |
       U3F | U4R | U4F ;

type thruster_list = [thruster_name];

data control_mode_switch = ROT | TRAN;
data AAH_control_button = button_up | button_down;

data AAH_engage_state = AAH_off | AAH_started | AAH_on | pressed_once
       | AAH_closing | pressed_twice;

click_timeout = (100 as nat 32);

naa true ZERO _ = true;
naa true _ true = true;
naa  _    _    _   = false;

incone n = n + (1 as nat 32);

comb_rot_cmds   ZERO  aah  _   = aah;
comb_rot_cmds    _    aah true = aah;
comb_rot_cmds hcm_rot aah  _   = hcm_rot;

thruster2string thruster =
    case thruster of
         B1 -> "B1"
       | B2 -> "B2"
       | B3 -> "B3"
       | B4 -> "B4"
       | F1 -> "F1"
       | F2 -> "F2"
       | F3 -> "F3"
       | F4 -> "F4"
```

```
       | L1R -> "L1R"
       | L1F -> "L1F"
       | R2R -> "R2R"
       | R2F -> "R2F"
       | L3R -> "L3R"
       | L3F -> "L3F"
       | R4R -> "R4R"
       | R4F -> "R4F"
       | D1R -> "D1R"
       | D1F -> "D1F"
       | D2R -> "D2R"
       | D2F -> "D2F"
       | U3R -> "U3R"
       | U3F -> "U3F"
       | U4R -> "U4R"
       | U4F -> "U4F";

thrusterlist2string []     = "\n";
thrusterlist2string (x:xs) = thruster2string x ++ " " ++ thrusterlist2string xs;

stream output to "std_out";

box sensors
  in  ( state :: nat 32 )
  out  ( state' :: nat 32,
         vert, horiz, trans, twist :: axis_command , mode :: control_mode_switch,
         button :: AAH_control_button)
match
  0 ->  (1,NEG,NEG,NEG,NEG,ROT,button_up)
| 1 ->  (2,NEG,NEG,ZERO,NEG,ROT,button_up)
| 2 ->  (0,POS,NEG,ZERO,NEG,ROT,button_down);

wire sensors
( sensors.state' initially 0)
( sensors.state, SAFER.vert, SAFER.horiz, SAFER.trans,
  SAFER.twist, SAFER.mode, SAFER.button);

box SAFER
  in  ( aa,ihcm::axis_pred,toggle::AAH_engage_state,tout,clock :: nat 32,
        vert, horiz, trans, twist :: axis_command,
        mode :: control_mode_switch, button :: AAH_control_button )
  out ( aa', ihcm'::axis_pred,toggle'::AAH_engage_state,tout',clock'::nat 32,
        thrusters::string )
match
  (_,_,_,_,_,_,_,_,_,_,_,_,_) -> (_,_,_,_,_,_,_)
boxes

  box grip_command
    in  ( vert, horiz, trans, twist :: axis_command , mode :: control_mode_switch )
    out ( tran, rot1, rot2 :: command )
  match
    (v,h,tr,tw,TRAN) -> ((h,tr,v),(ZERO,tw,ZERO),(ZERO,tw,ZERO))
  | (v,h,tr,tw,ROT)  -> ((h,ZERO,ZERO),(v,tw,tr),(v,tw,tr));

  wire grip_command
     ( SAFER.vert, SAFER.horiz, SAFER.trans, SAFER.twist, SAFER.mode )
```

```
   ( integrated_commands.tran, integrated_commands.rot, AAH_transition.rot );

box AAH_pre_fanout
  in ( aa::axis_pred,toggle::AAH_engage_state,tout,clock::nat 32,ihcm::axis_pred)
  out (aa1,aa2::axis_pred,aa3::axis_pred,ichm1::axis_pred,toggle1,toggle2::
       AAH_engage_state,tout1,tout2,clock1,clock2::nat 32,ihcm2::axis_red)
match
 (aa,toggle,tout,clo,ichm) -> (aa,aa,aa,ichm,toggle,toggle,tout,tout,clo,clo,ihcm);

wire AAH_pre_fanout
  ( SAFER.aa,SAFER.toggle,SAFER.tout,SAFER.clock, SAFER.ihcm )
  ( AAH_button_transition.aa, AAH_transition.aa,
    integrated_commands.aa,integrated_commands.ihcm,
    AAH_button_transition.state, AAH_transition.toggle,
    AAH_button_transition.tout, AAH_transition.tout,
    AAH_button_transition.clock, AAH_transition.clock,AAH_transition.ihcm);

box AAH_button_transition
  in  ( state :: AAH_engage_state, button :: AAH_control_button,
        aa :: axis_pred, clock, tout :: nat 64 )
  out ( state', stateout :: AAH_engage_state )
match
  (AAH_off,button_down,_,_,_)                    -> (AAH_started,AAH_started)
| (AAH_started,button_down,_,_,_)                -> (AAH_started,AAH_started)
| (AAH_on,button_down,_,_,_)                     -> (pressed_once,pressed_once)
| (pressed_once,button_down,_,_,_)               -> (pressed_once,pressed_once)
| (AAH_closing,button_down,_,_,_)                -> (pressed_twice,pressed_twice)
| (pressed_twice,button_down,_,_,_)              -> (pressed_twice,pressed_twice)
| (AAH_off,button_up,_,_,_)                      -> (AAH_off,AAH_off)
| (AAH_started,button_up,_,_,_)                  -> (AAH_on,AAH_on)
| (AAH_on,button_up,(false,false,false),_,_)     -> (AAH_off,AAH_off)
| (AAH_on,button_up,_,_,_)                       -> (AAH_on,AAH_on)
| (pressed_once,button_up,_,_,_)                 -> (AAH_closing,AAH_closing)
| (AAH_closing,button_up,(false,false,false),_,_) -> (AAH_off,AAH_off)
| (AAH_closing,button_up,_,cl,tout)              -> if cl > tout
                                                   then (AAH_on,AAH_on)
                                                   else (AAH_closing,AAH_closing)
| (pressed_twice,button_up,(_,_,_),_,_)         -> (AAH_off,AAH_off);

wire AAH_button_transition
( AAH_pre_fanout.toggle1,
  SAFER.button, AAH_pre_fanout.aa1,
  AAH_pre_fanout.clock1, AAH_pre_fanout.tout1)
( SAFER.toggle', AAH_transition.engage);

box AAH_transition
  in (  aa :: axis_pred, engage, toggle :: AAH_engage_state,
        rot :: command, ihcm :: axis_pred,
        clock, tout :: nat 64 )
  out ( aa', ihcm' :: axis_pred,
        clock', timeout' :: nat 64 )
match
  (_,AAH_off,AAH_started,(r1,r2,r3),_,cl,tout)
          -> ((true,true,true),(r1 != ZERO,r2 != ZERO,r3 != ZERO),incone cl,tout)
| (_,_,AAH_off,_,ihcm,cl,tout)
          -> ((false,false,false),ihcm,incone cl,tout)
```

```
| ((a1,a2,a3),AAH_on,pressed_once,(r1,r2,r3),(i1,i2,i3),cl,tout)
           -> ((naa a1 r1 i1, naa a2 r2 i2,
                naa a3 r3 i3),(i1,i2,i3),incone cl,cl+click_timeout)
| ((a1,a2,a3),eng,_,(r1,r2,r3),(i1,i2,i3),cl,tout)
           -> ((naa a1 r1 i1,naa a2 r2 i2,
                naa a3 r3 i3),(i1,i2,i3),incone cl,tout);

wire AAH_transition
   ( AAH_pre_fanout.aa2, -- active axis
     AAH_button_transition.stateout, AAH_pre_fanout.toggle2,
     grip_command.rot2, -- grip command
     AAH_pre_fanout.ihcm2, -- ignore_hcm
     AAH_pre_fanout.clock2, AAH_pre_fanout.tout2 )
   ( AAH_post_fanout.aa, -- active axis
     AAH_post_fanout.ihcm, -- ignore_hcm
     SAFER.clock', SAFER.tout' );

box AAH_post_fanout
  in ( aa, ihcm :: axis_pred)
  out ( aah :: command,
        aa1, ihcm1 :: axis_pred )
match
  (aa,ihcm) -> ((ZERO,ZERO,ZERO),aa,ihcm);

wire AAH_post_fanout
  ( AAH_transition.aa', AAH_transition.ihcm' )
  ( integrated_commands.aah,
    SAFER.aa',SAFER.ihcm');

box integrated_commands
  in  ( tran,rot,aah::command,aa,ihcm::axis_pred )
  out ( rot1,rot2::command,prio::bool )
match
  (tran,(ZERO,ZERO,ZERO),_,(false,false,false),_)
     -> (tran,(ZERO,ZERO,ZERO),true)
| (_,(r,p,ya),_,(false,false,false),_)
     -> ((ZERO,ZERO,ZERO),(r,p,ya),false)
| (tran,(ZERO,ZERO,ZERO),(a1,a2,a3),_,_)
    -> (tran,(a1,a2,a3),true)
| (_,(r,p,ya),(a1,a2,a3),_,(i1,i2,i3))
    -> ((ZERO,ZERO,ZERO),
        (comb_rot_cmds r a1 i1,comb_rot_cmds p a2 i2,comb_rot_cmds ya a3 i3),
        false);

wire integrated_commands
  ( grip_command.tran, grip_command.rot1, AAH_post_fanout.aah,
    AAH_pre_fanout.aa3, AAH_pre_fanout.ihcm2 )
  ( prioritized_tran_cmd.tran, integrated_fanout.rot,
    prioritized_tran_cmd.isPrio );

box prioritized_tran_cmd
  in  ( tran :: command, isPrio :: bool)
  out ( x', y', z' :: axis_command )
match
  (tran,false)         ->       tran
| ((ZERO,ZERO,ZERO),_) -> (ZERO,ZERO,ZERO)
```

```
| ((ZERO,ZERO,zacc),_) -> (ZERO,ZERO,zacc)
| ((ZERO,yacc,_),_)    -> (ZERO,yacc,ZERO)
| ((xacc,_,_),_)       -> (xacc,ZERO,ZERO) ;

wire prioritized_tran_cmd
 ( integrated_commands.rot1, integrated_commands.prio )
 ( BF.A, LRUD.A, LRUD.B );

box integrated_fanout
  in  (rot :: command)
  out (p', y', r' :: axis_command, rot' :: command)
match
  ((p,y,r)) -> (p,y,r,(p,y,r));

wire integrated_fanout
 ( integrated_commands.rot2 )
 ( BF.B, BF.C, LRUD.C, thruster_join.rot );

box LRUD
  in  ( A, B, C :: axis_command )
  out ( M, O :: thruster_list )
match
     (NEG,NEG,NEG)   -> ([],[])
   | (NEG,NEG,ZERO)  -> ([],[])
   | (NEG,NEG,POS)   -> ([],[])
   | (NEG,ZERO,NEG)  -> ([L1R],[L1F,L3F])
   | (NEG,ZERO,ZERO) -> ([L1R,L3R],[L1F,L3F])
   | (NEG,ZERO,POS)  -> ([L3R],[L1F,L3F])
   | (NEG,POS,NEG)   -> ([],[])
   | (NEG,POS,ZERO)  -> ([],[])
   | (NEG,POS,POS)   -> ([],[])
   | (ZERO,NEG,NEG)  -> ([U3R],[U3F,U4F])
   | (ZERO,NEG,ZERO) -> ([U3R,U4R],[U3F,U4F])
   | (ZERO,NEG,POS)  -> ([U4R],[U3F,U4F])
   | (ZERO,ZERO,NEG) -> ([L1R,R4R],[])
   | (ZERO,ZERO,ZERO)-> ([],[])
   | (ZERO,ZERO,POS) -> ([R2R,L3R],[])
   | (ZERO,POS,NEG)  -> ([D2R],[D1F,D2F])
   | (ZERO,POS,ZERO) -> ([D1R,D2R],[D1F,D2F])
   | (ZERO,POS,POS)  -> ([D1R],[D1F,D2F])
   | (POS,NEG,NEG)   -> ([],[])
   | (POS,NEG,ZERO)  -> ([],[])
   | (POS,NEG,POS)   -> ([],[])
   | (POS,ZERO,NEG)  -> ([R4R],[R2F,R4F])
   | (POS,ZERO,ZERO) -> ([R2R,R4R],[R2F,R4F])
   | (POS,ZERO,POS)  -> ([R2R],[R2F,R4F])
   | (POS,POS,NEG)   -> ([],[])
   | (POS,POS,ZERO)  -> ([],[])
   | (POS,POS,POS)   -> ([],[]);

wire LRUD
  ( prioritized_tran_cmd.y', prioritized_tran_cmd.z', integrated_fanout.r' )
  ( thruster_join.lm, thruster_join.lo );

box BF
  in  ( A, B, C :: axis_command )
```

```
    out ( M, O :: thruster_list )
  match
       (NEG,NEG,NEG)   -> ([B4],[B2,B3])
     | (NEG,NEG,ZERO)  -> ([B3,B4],[])
     | (NEG,NEG,POS)   -> ([B3],[B1,B4])
     | (NEG,ZERO,NEG)  -> ([B2,B4],[])
     | (NEG,ZERO,ZERO) -> ([B1,B4],[B2,B3])
     | (NEG,ZERO,POS)  -> ([B1,B3],[])
     | (NEG,POS,NEG)   -> ([B2],[B1,B4])
     | (NEG,POS,ZERO)  -> ([B1,B2],[])
     | (NEG,POS,POS)   -> ([B1],[B2,B3])
     | (ZERO,NEG,NEG)  -> ([B4,F1],[])
     | (ZERO,NEG,ZERO) -> ([B4,F2],[])
     | (ZERO,NEG,POS)  -> ([B3,F2],[])
     | (ZERO,ZERO,NEG) -> ([B2,F1],[])
     | (ZERO,ZERO,ZERO)-> ([],[])
     | (ZERO,ZERO,POS) -> ([B3,F4],[])
     | (ZERO,POS,NEG)  -> ([B2,F3],[])
     | (ZERO,POS,ZERO) -> ([B1,F3],[])
     | (ZERO,POS,POS)  -> ([B1,F4],[])
     | (POS,NEG,NEG)   -> ([F1],[F2,F3])
     | (POS,NEG,ZERO)  -> ([F1,F2],[])
     | (POS,NEG,POS)   -> ([F2],[F1,F4])
     | (POS,ZERO,NEG)  -> ([F1,F3],[])
     | (POS,ZERO,ZERO) -> ([F2,F3],[F1,F4])
     | (POS,ZERO,POS)  -> ([F2,F4],[])
     | (POS,POS,NEG)   -> ([F3],[F1,F4])
     | (POS,POS,ZERO)  -> ([F3,F4],[])
     | (POS,POS,POS)   -> ([F4],[F2,F3]);

  wire BF
    ( prioritized_tran_cmd.x', integrated_fanout.p', integrated_fanout.y' )
    ( thruster_join.bm, thruster_join.bo );

  box thruster_join
    in  ( lm, lo, bm, bo :: thruster_list, rot :: command )
    out ( thr :: string)
  match
    (lm,lo,bm,bo,(ZERO,ZERO,ZERO)) -> thrusterlist2string (lm ++ lo ++ bm ++ bo)
  | (lm,lo,bm,_,(ZERO,ZERO,_))     -> thrusterlist2string (lm ++ lo ++ bm)
  | (lm,_,bm,bo,(_,_,ZERO))        -> thrusterlist2string (lm ++ bm ++ bo)
  | (lm,_,bm,_,(_,_,_))            -> thrusterlist2string (lm ++ bm ) ;

  wire thruster_join
    ( LRUD.M, LRUD.O, BF.M, BF.O, integrated_fanout.rot' )
    ( SAFER.thrusters );
end;

wire SAFER ( SAFER.aa' initially (false,false,false),
             SAFER.ihcm' initially (false,false,false),
             SAFER.toggle' initially AAH_off,
             SAFER.tout' initially 0 as nat 32,
             SAFER.clock' initially 0 as nat 32,
             sensors.vert , sensors.horiz, sensors.trans, sensors.twist, sensors.mode,
             sensors.button)
           ( SAFER.aa,SAFER.ihcm,SAFER.toggle,SAFER.tout,SAFER.clock,output );
```

Bibliography

[1] ISO/IEC 15437:2001. Information Technology – E-LOTOS, 2001. ISO/IEC International Standard.

[2] Martín Abadi. An Axiomatization of Lamport's Temporal Logic of Actions. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 57–69, Amsterdam, The Netherlands, August 1990. Springer-Verlag.

[3] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, 31 May 1991.

[4] Martín Abadi and Leslie Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[5] Martín Abadi and Leslie Lamport. Conjoining Specifications. *j-TOPLAS*, 17(3):507–534, May 1995.

[6] Martín Abadi and Stephan Merz. An abstract account of composition. In J. Wiedermann and P. Hajek, editors, *Mathematical Foundations of Computer Science*, volume 969 of *Lecture Notes in Computer Science*, pages 499–508, Prague, Czech Republic, 1995. Springer-Verlag.

[7] Martín Abadi and Stephan Merz. On TLA as a Logic. In Manfred Broy, editor, *Deductive Program Design*, NATO ASI series F, pages 235–272. Springer-Verlag, Berlin, 1996.

[8] Jean-Raymond Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[9] Jean-Raymond Abrial. *Modelling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. To be published.

[10] Jean-Raymond Abrial and Louis Mussat. Introducing dynamic constraints in B. In Didier Bert, editor, *B'98 : The 2nd International B Conference*, volume 1393 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 83–128, Montpellier, April 1998. B1998, LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer Verlag.

[11] Sten Agerholm. A HOL Basis for Reasoning about Functional Programs. BRICS RS-94-44, ISSN 0909-0878, Department of Computer Science, University of Aarhus, Denmark, December 1994. http://www.daimi.aau.dk/BRICS/RS/94/44/BRICS-RS-94-44/BRICS-RS-94-44.html.

[12] Sten Agerholm and Peter Gorm Larsen. Modeling and Validating SAFER in VDM-SL. In Michael Holloway, editor, *Fourth NASA Langley Formal Methods Workshop*, September 1997.

[13] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21:181–185, October 1985.

[14] Peter Amey. Correctness by Construction: Better Can Also be Cheaper. *CrossTalk - The Journal of Defense Software Engineering*, March 2002.

[15] Hasan Amjad. Combining Model Checking and Theorem Proving. Technical Report UCAM-CL-TR-601, University of Cambridge Computer Laboratory, 2004.

[16] David Aspinall. Proof General: A Generic Tool for Proof Development. In Susanne Graf and Michael I. Schwartzbach, editors, *In Proceedings of 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00), Held as Part of the European Joint Conferences on the Theory and Practice of Software (ETAPS'00), Berlin, Germany*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, April 2000.

[17] Serge Autexier, Dieter Hutter, Bruno Langenstein, Heiko Mantel, Georg Rock, Axel Schairer, Werner Stephan, Roland Vogt, and Andreas Wolpers. VSE: Formal Methods Meet Industrial Needs. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(1):66–77, September 2000.

[18] J. C. M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Scisence*, 335(2-3):131–146, 2005.

[19] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Conference Record of POPL'02: The 29th ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 1–3, Portland, Oregon, 2002.

[20] Clemens Ballarin. Locales and Locale Expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.

[21] Clemens Ballarin. Interpretation of Locales in Isabelle: Theories and Proof Contexts. In Jonathan M. Borwein and William M. Farmer, editors, *Mathematical Knowledge Management, 5th International Conference, MKM 2006, Wokingham, UK*, volume 4108 of *Lecture Notes in Computer Science*, pages 31–43. Springer, August 2006.

[22] Henk P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[23] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[24] Brannon Batson and Leslie Lamport. High-Level Specifications: Lessons from Industry. In *Formal Methods for Components and Objects*, number 2852 in Lecture Notes in Computer Science, pages 242–261. Springer, March 17 2003.

[25] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[26] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[27] Nikolaj Bjorner, Anca Browne, Michael Colon, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomas Uribe. *Verifying Temporal Properties of Reactive Systems: A STeP Tutorial*. Kluwer Academic Publishers, 2000.

[28] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, Cambridge, England, 2001.

[29] Armelle Bonenfant, Christian Ferdinand, Kevin Hammond, Reinhold Armelle Bonenfant, Christian Ferdinand, Kevin Hammond, and Reinhold Heckmann. Worst-case execution times for a purely functional language. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Proceedings Implementation of Functional Languages (IFL 2006)*, volume 4449 of *Lecture Notes in Computer Science*, pages 235 – 252. Springer, 2006.

[30] Edwin Brady, James Hugh McKinna, and Kevin Hammond. Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types. In Marco T. Morazan, editor, *Trends in Functional Programming*, volume 8, chapter 10, pages 159 – 176. Intellect, 2007.

[31] Y. Brave and M. Heymann. Control of Discrete Event Systems Modeled as Hierarchical State Machines. In *Proceedings of 30th IEEE Conference on Decision and Control*, pages 1499–1504, Brighton, UK, December 1991.

[32] Alan Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In *9th International Conference on Automated Deduction (CADE'09)*, pages 111–20. Springer-Verlag, 1987.

[33] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling – Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[34] Alan Bundy, Alan Smaill, and Jane Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proceeding of UK IT 90*, pages 221–6. IEE, 1990.

[35] Rodney Martineau Burstall. Program proving as hand simulation with a little induction. In *International congress of the International Federation for Information Processing (IFIP '74)*, pages 308–312. Elsevier/North-Holland, 1974.

[36] Rodney Martineau Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[37] Michael J. Butler. csp2B: A Practical Approach to Combining CSP and B. *Formal Aspect of Computing*, 12(3):182–198, 2000.

[38] Michael J. Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2005.

[39] Michael J. Butler and Divakar Yadav. An Incremental Development of the Mondex System in Event-B. *Formal Aspect of Computing*, 20(1):61–77, 2008.

[40] Ana Cavalcanti and Jim Woodcock. ZRC - A refinement calculus for Z. *Formal Aspect of Computing*, 10(3):267–289, 1998.

[41] Alonzo Church and John Barkley Rosser. Some Properties of Conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[42] Edmund M. Clarke and Ernest Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[43] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[44] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. Part of the SEI Series in Software Engineering series.

[45] William F. Clocksin and Christopher S. Mellish. *Programming in PROLOG*. Springer-Verlag, SPRIad, 1984.

[46] Andrew Cook, Andrew Ireland, Greg J. Michaelson, and Norman Scaife. Discovering applications of higher order functions through proof planning. *Journal of Formal Aspects of Computing*, 17(1):38–57, 2005.

[47] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. *SIGPLAN Not.*, 42(1):265–276, 2007.

[48] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. *ACM SIGPLAN Notices*, 41(6):415–426, June 2006.

[49] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond Safety. In *International Conference on Computer-Aided Verification*, pages 415–418, Seattle, 2006.

[50] Catarina Coquand and Thierry Coquand. Structured Type Theory, June 1999. Preliminary version.

[51] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[52] Satyaki Das. *Predicate Abstraction*. PhD thesis, Stanford University, December 2005.

[53] Marco Devillers, David Griffioen, and Olaf Müller. Possibly Infinite Sequences in Theorem Provers: A comparative study. In Elsa L. Gunter and Amy P. Felty, editors, *10th International Conference on Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 89–104. Springer, August 1997.

[54] Raymond Devillers, Hanna Klaudel, and Robert C. Riemann. General Parameterised Refinement and Recursion for the M-net Calculus. *Theoretical Computer Science*, 300(1-3):259–300, May 2003.

[55] DevTopics. 20 Famous Software Disasters – Part 2. http://www.devtopics.com/20-famous-software-disasters-part-2. November 2008.

[56] Edsger Wybe Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. A.C.M.*, 18(8):453–457, August 1975.

[57] Lucas Dixon. *A Proof Planning Framework for Isabelle.* PhD thesis, University of Edinburgh, 2005.

[58] Lucas Dixon and Jaques Fleuriot. IsaPlanner: A Prototype Proof Planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.

[59] Lucas Dixon and Jaques Fleuriot. Higher order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 83–98, 2004.

[60] Jack Dongarra, Ian Foster, Geoffrey C. Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors. *The Sourcebook of Parallel Computing.* Morgan Kufmann, San Francisco, CA, USA, 2002.

[61] Doron Drusinsky and David Harel. Using Statecharts for Hardware Description and Synthesis. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 8(7):798–807, 1989.

[62] Matthew B. Dwyer, Corina S. Pasareanu, and James C. Corbett. Translating Ada Programs for Model Checking : A Tutorial. Technical Report KSU CIS TR-98-12, Kansas State University, 1998.

[63] Sidi O. Ehmety and Lawrence C. Paulson. Representing Component States in Higher-Order Logic. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, pages 151–158, 2001.

[64] EmBounded Homepage. December 2008. http://www.embounded.org. December 2008.

[65] Urban Engberg, Peter Gronning, and Leslie Lamport. Mechanical Verification of Concurrent Systems with TLA. In Gregor von Bochmann and David K. Probst, editors, *Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55. Springer, 1992.

[66] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, Université Paris Sud, 2003.

[67] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer-Verlag, 2004.

[68] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[69] Clemens Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[70] Robert W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.

[71] Cedric Fournet and Georges Gonthier. The Join Calculus: a Language for Distributed Mobile Programming. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer, 2000.

[72] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, August 1999.

[73] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

[74] Simson Garfinkel. History's Worst Software Bugs. Wired.com, August 2005. http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all. November 2008.

[75] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation, Systems Research Center, November 1991.

[76] Gerhard Gentzen. Untersuchungen über das logisches Schließen. *Mathematische Zeitschrift*, 1:176–210, 1935.

[77] Arkadeb Ghosal, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan. A Hierarchical Coordination Language for Interacting Real-Time Tasks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, pages 132–141, New York, NY, USA, 2006. ACM.

[78] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatsh. Math. Phys.*, 38:173–198, 1931.

[79] Herman Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. In A. Traub, editor, *Collection Works of J. von Neumann*, pages 80–151. Pergamon, 1949. Originally, a report of the U.S. Ordinance Department.

[80] Michael J. Gordon. HOL: A proof generating system for Higher-Order Logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[81] Michael J. Gordon. *Programming Language Theory and its Implementation*. International Series in Computer Science. Prentice Hall, 1988.

[82] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[83] M. Griffiths. Program Production by Successive Transformation. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 125–152. Springer, 1975.

[84] Gudmund Grov. Model Checking HW-Hume. Master's thesis, Heriot-Watt University, 2004.

[85] Gudmund Grov. Verifying the Correctness of Hume Program - An Approach Combining Algorithmic and Deductive Reasoning. In *Proceedings of the* 20^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE-05)*, pages 444–447. ACM Press, 2005.

[86] Gudmund Grov and Andrew Ireland. Towards Automated Property Discovery within Hume. In Andrew Ireland and Laura Kovacs, editors, *2nd International Workshop on Invariant Generation (WING'09)*, pages 45–59, 2009.

[87] Gudmund Grov and Greg Michaelson. Towards a Box Calculus for Hierarchical Hume. In Marco T. Morazan, editor, *Trends in Functional Programming*, volume 8, chapter 5, pages 71 – 88. Intellect, 2007.

[88] Gudmund Grov, Greg Michaelson, and Andrew Ireland. Formal Verification of Concurrent Scheduling Strategies using TLA. In *3rd IEEE International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, number CFP07036-USB in IEEE Catalog Number. IEEE, 2007.

[89] Gudmund Grov, Robert Pointon, Greg Michaelson, and Andrew Ireland. Preserving Coordination Properties when Transforming Concurrent System Components. In *Coordination Models, Languages and Applications Track of the 23rd Annual ACM Symposium on Applied Computing*, volume 1 of 3, pages 126 – 127, 1515 Broadway, New York, March 2008. The Association for Computing Machinery, Inc.

[90] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hofman, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards Formally Verifiable WCET Analysis for a Functional Programming Language. In *Proceedings of 6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[91] Kevin Hammond, Gudmund Grov, Greg Michaelson, and Andrew Ireland. Low-level programming in Hume: an exploration of the HW-Hume level. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *In Proceedings of Implementation of Functional Languages (IFL 2006)*, volume 4449 of *Lecture Notes in Computer Science*, pages 91 – 107. Springer, 2006.

[92] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Second International*

Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings, volume 2830 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2003.

[93] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[94] Jaques Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, University of Paris, 1930.

[95] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[96] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[97] Gerard J. Holzmann. *The Spin Model Checker — Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[98] Hume Homepage. http://www.hume-lang.org. December 2008.

[99] Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In H-W. Loidl, editor, *Trends in Functional Programming*, volume 5, pages 49–64, 2006.

[100] Alexei Iliasov. *Design Components*. PhD thesis, Newcastle University, July 2008.

[101] Gartner Inc. http://www.gartner.com. November 2008.

[102] Andrew Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 of *LNAI*, pages 178–189, St. Petersburg, Russia, July 1992. Springer Verlag.

[103] Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *JAR: Journal of Automated Reasoning*, 16:79 – 111, 1996.

[104] Andrew Ireland, Bill J. Ellis, Andrew Cook, Rod Chapman, and John Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

[105] Andrew Ireland and Jamie Stark. On the Automatic Discovery of Loop Invariants. In *The Fourth NASA Langley Formal Methods Workshop*, number 3356. NASA Conference Publication, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

[106] Andrew Ireland and Jamie Stark. Proof Planning for Strategy Development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001.

[107] Cliff B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.

[108] Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, 1986.

[109] Cliff B. Jones. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[110] Sam Jones, Gillian Tett, and Paul J. Davies. Moody's error gave top ratings to debt products. Financial Times, May 2008. http://us.ft.com/ftgateway/superpage.ft?news_id=fto052020081848170760. November 2008.

[111] Simon Peyton Jones. *Haskell 98 Language and Libraries – the Revised Report*. CUP, April 2003.

[112] Steffen Jost. Formal Hume Semantics. EmBounded Project Deliverable, 2008. Deliverable D12. Availabe at http://www.embounded.org/.

[113] Andreas Jungmaier. Translation of TLA-Specifications to Ada. Unpublished, University of Cantabria (Spain) and University of Duisburg (Germany), September 1998.

[114] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The Definition of Extended ML: A Gentle Introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[115] Sara Kalvala. A Formulation of TLA in Isabelle. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 214–228, Aspen Grove, Utah, USA, September 1995. Springer-Verlag.

[116] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - A Sectioning Concept for Isabelle. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 1999.

[117] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[118] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2), 1977.

[119] Leslie Lamport. *win* and *sin*: Predicate Transformers for Concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3):396–428, July 1990.

[120] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[121] Leslie Lamport. Adding "Process Algebra" to TLA. Unpublished note, January 1995.

[122] Leslie Lamport. How to Write a Proof. *American Mathematical Monthly*, 102(7):600–608, 1995.

[123] Leslie Lamport. What Process Algebra Proofs Use Instead of Invariance. Unpublished note, January 1995.

[124] Leslie Lamport. *Specifying Systems — The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Reading, Massachusetts, 2002.

[125] Leslie Lamport. Real-Time Model Checking Is Really Simple. In Dominique Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2005.

[126] Leslie Lamport. *TLA*$^{+2}$ *– A Prelimenary Guide*, April 2008. http://research.microsoft.com/users/lamport/tla/tools.html.

[127] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Transactions on Programming Langauges and Systems*, 21(3):502–526, May 1999.

[128] Thomas Langbacka. A HOL Formalisation of the Temporal Logic of Actions. In Thomas F. Melham and Juanito Camilleri, editors, *Iinternational Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 332–345, Valletta, Malta, September 1994. Springer.

[129] John Langley and Randy Pollack. Reasoning About CBV Functional Programs in Isabelle/HOL. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 201–216, 2004.

[130] P. G. Larsen, B. S. Hansen, H. Brunn N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information Technology – Programming Languages, their Environments and System Software Interfaces – Vienna Development Method – Specification Language – Part 1: Base Language, December 1996.

[131] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[132] Dirk Leinenbach and Elena Petrova. Pervasive Compiler Verification – From Verified Programs to Verified Systems. *Electrical Notes on Theoretical Computer Science*, 217:23–40, 2008.

[133] Michael Leuschel and Michael Butler. ProB: an Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer*, 10(2):185–203, March 2008.

[134] Huiqing Li and Simon Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In *Proceedings of 6th IEEE Workshop on Source Code Analysis and Manipulation, Philadelphia, USA*, pages 197–206, September 2006.

[135] Hans-Wolfgang Loidl and Gudmund Grov. Assertion Language. EmBounded Project Deliverable, October 2007. Deliverable D17. Available at `http://www.embounded.org/`.

[136] Peter Madden. *Automated Program Transformation Through Proof Transformation*. PhD thesis, University of Edinburgh, 1991.

[137] Zohar Manna. *Mathematical Theory of Computing*. McGraw-Hill, 1974.

[138] Claude Marche, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[139] James Hugh McKinna. *Deliverables: a categorical approach to program development in type theory*. PhD thesis, University of Edinburgh, 1992.

[140] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[141] Erica Melis and Jörg Siekmann. Knowledge-based proof planning. 115(1):65–105, 1999.

[142] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 372–384. Springer, 2004.

[143] Stephan Merz. An Encoding of TLA in Isabelle. http://www.pst.informatik.uni-muenchen.de/~merz/isabelle/, 1998.

[144] Stephan Merz. A More Complete TLA. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1226–1244, Toulouse, France, September 1999. Springer-Verlag.

[145] Stephan Merz. On the Logic of TLA+. *Computers and Informatics – Special Issue on the semantics of specification formalisms*, 22:351 – 379, 2003.

[146] Greg Michaelson, Kevin Hammond, and Jocelyn Sérot. FSM-Hume: Programming Resource-Limited Systems using Bounded Automata. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1455–1461. ACM Press, March 2004.

[147] Ivana Mijajlovic and Noah Torp-Smith. Refinement in Separation Context. In *In Proceedings of the 2nd workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'04)*, Venice, Italy, January 2004.

[148] Ivana Mijajlovic, Noah Torp-Smith, and Peter W. O'Hearn. Refinement and Separation Contexts. In Kamal Lodaya and Meena Mahajan, editors, *In Proceedings of 24th International Conference on Foundations of Software Technology and Theoretical Computer Science, Chennai, India*, volume 3328 of *Lecture Notes in Computer Science*, pages 421–433. Springer, December 2004.

[149] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[150] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[151] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[152] Ministry of Defence. The Procurement of Safety Critical Software in Defence Equipment (Part 1: Requirements, Part 2: Guidance). Defence Standard 00-55, Issue 1, Ministry of Defence, Directorate of Standardization, 1991.

[153] NASA. Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems Volume II: A Practitioner's Companion, 1997.

[154] Georg Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[155] Allen Newell and Herbert A. Simon. The Logic Theory Machine. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill, 1963.

[156] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[157] Tobias Nipkow. Structured proofs in isar/HOL. In Herman Geuvers and Freek Wiedijk, editors, *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 259–278. Springer, 2002.

[158] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[159] Tobias Nipkow and Leonor Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577, pages 188–203, 1999.

[160] Thomas Noll, Lars Ake Fredlund, and Dilian Gurov. The Erlang Verification Tool. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 582–585. Springer–Verlag, 2001.

[161] Peter W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[162] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[163] Susan Owicki and David Gries. Verifying Propeties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–284, May 1976.

[164] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, pages 748–752, Saratoga, New-York, 1992. Springer-Verlag.

[165] Lawrence C. Paulson. Designing a Theorem Prover. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 415–475. Oxford University Press, 1992.

[166] Lawrence C. Paulson. Organizing Numerical Theories Using Axiomatic Type Classes. *Journal Automated Reasoning*, 33(1):29–49, 2004.

[167] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[168] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[169] Patrick Regan and Scott Hamilton. NASA's Mission Reliable. *Innovative Technology for Computer Professionals: Computer*, pages 59–68, January 2004.

[170] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science*, pages 55–74, Los Alamitos, CA, USA, July 2002. IEEE Computer Society.

[171] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: A Flexible Framework for Creating Software Model Checkers. In Phil McMinn, editor, *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006), 29-31 August 2006, Windsor, United Kingdom*, pages 3–22. IEEE Computer Society, 2006.

[172] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[173] Georg Rock, Werner Stephan, and Andreas Wolphers. Modular Reasoning about Structured TLA Specifications. In Rudolf In Berghammer and Yassine Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 217–229. Springer Verlag, 1999.

[174] John Rushby. Formal Methods and the Certification of Critical Systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title "Formal Methods and Digital Systems Validation for Airborne Systems" as NASA Contractor Report 4551, December 1993.

[175] Norman Scaife, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, and Jocelyn Serot. Costing by Construction: Compositional Design and Verification of Embedded Applications using the Hume Software Development Methodology, 2008. In preparation. Available from http://www.hume-lang.org.

[176] Norbert Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2004.

[177] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[178] Steve Schneider and Helen Treharne. CSP Theorems for Communicating B Machines. *Formal Aspects of Computing: Applicable Formal Methods*, 17(4):390–422, Dec 2005.

[179] Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[180] Neil Storey. *Safety Critical Computer Systems*. Addison-Wesley, 1996.

[181] Li Tan. Model-Based Self-Adaptive Embedded Programs with Temporal Logic Specifications. In *QSIC '06: Proceedings of the Sixth International Conference on Quality Software*, pages 151–158, Washington, DC, USA, 2006. IEEE Computer Society.

[182] Alfred Tarski. Ueber einige fundamentale begriffe der metamathematik. *Comptes Rendus de Séances de la Société des Sciences et des Lettres de Varsovie, Classe III*, 23:22–29, 1930.

[183] Simon Thompson. Refactoring Functional Programs. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 331–357. Springer Verlag, September 2005.

[184] Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, The University of New South Wales, 2008.

[185] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society (2)*, 42:230–265, 1936-7.

[186] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines, EDSAC Inaugural Conference*, pages 67–69, 1949.

[187] Christian Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automatic Reasoning*, 40(4):327–356, May 2008.

[188] Christian Urban, Julien Narboux, and Stefan Berghofer. The Nominal Datatype Package. Notes on the nominal datatype in Isabelle. Available from http://isabelle.in.tum.de/nominal/download.html.

[189] Pedro Baltazar Vasconcelos. *Space Costing Analysis Using Sized Types*. PhD thesis, University of St. Andrews, 2008.

[190] Verisoft Homepage. http://www.verisoft.de. December 2008.

[191] Eelco Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, May 2001.

[192] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[193] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2):1–36, April 2003.

[194] Ben L. Di Vito. High-Automation Proofs for Properties of Requirements Models. *International Journal on Software Tools for Technology Transfer*, 3(1):20–31, 2000.

[195] Philip Wadler. The Essence of Functional Programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, New York, NY, USA, 1992. ACM.

[196] Philip Wadler and Stephan Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[197] Matthew Wahab. The Semantics of TLA on the PVS Theorem Prover. Research Report CS-RR-317, Department of Computer Science, University of Warwick, Coventry, UK, October 1996.

[198] Tjark Weber. Towards Mechanized Program Verification with Separation Logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004.

[199] Markus Wenzel. Using Axiomatic Type Classes in Isabelle, May 2000.

[200] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2002.

[201] Martin Wildmoser and Tobias Nipkow. Certifying Machine Code Safety: Shallow versus Deep Embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305–320. Springer, 2004.

[202] Joakim Von Wright and Thomas Langbacka. Using a Theorem Prover for Reasoning About Concurrent Algorithms. In Gregor von Bochmann and David K. Probst, editors, *Proceedings of 4th International Computer Aided Verification Conference*, volume 663 of *Lecture Notes in Computer Science*, pages 56–68, 1992.

[203] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.